conference

proceedings

CARDIS '02
Fifth Smart Card Research and Advanced Application Conference

San Jose, California, USA
November 21–22, 2002

Sponsored by

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

and

IFIP WG 8.8

USENIX CARDIS '02: Fifth Smart Card Research and Advanced Application Conference *San Jose, CA Nov. 21–22, 2002*

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Phone: 510 528 8649
FAX: 510 548 5738
Email: office@usenix.org
URL: http://www.usenix.org

The price is $24 for members and $30 for nonmembers.
Outside the U.S.A. and Canada, please add
$12 per copy for postage (via air printed matter).

ISBN 1-931971-04-8

Printed in the United States of America on 50% recycled paper, 10–15% post-consumer waste.

USENIX Association
and IFIP WG 8.8 (Smart Cards)

Proceedings of

CARDIS '02

Fifth Smart Card Research and

Advanced Application Conference

November 21–22, 2002
San Jose, California, USA

Symposium Organizers

Program Chair

Peter Honeyman, *CITI, University of Michigan*

Program Committee

Isabelle Attali, *INRIA, Sophia Antipolis*
Boris Balacheff, *Hewlett Packard Labs, Bristol*
Yves Deswarte, *LAAS-CNRS, Toulouse*
Josep Domingo-Ferrer, *Universitat Rovira i Virgili, Tarragona*
Pieter Hartel, *Department of Computer Science, University of Twente*
Dirk Husemann, *IBM Research, Zurich*
Sebastien Jean, *LIFL, Lille*
Thomas Jensen, *IRISA/CNRS, Rennes*
Roger Kilian-Kehr, *T-Systems Nova GmbH, Darmstadt*
Marc Joye, *Gemplus, Gemenos*
Xavier Leroy, *INRIA, Rocquencourt, and Trusted Logic, Versailles*
Bernd Meyer, *Siemens AG, München*
Mike Montgomery, *SchlumbergerSema, Austin*
Joachim Posegga, *SAP Corporate Research, Karlsruhe*
Ton van der Putte, *Atos Origin, Utrecht*
Jean-Jacques Quisquater, *UCL, Louvain-la-Neuve*
Rüdiger Weis, *Vrije Universitat, Amsterdam*

The USENIX Association Staff

External Reviewers

Asker Bazen
Olivier Benoit
Andreas Bogk
Liqun Chen
Jordan C N Chong
Arnaud Contes
Ricardo Corin
Jean-Sébastien Coron
Benoit Gonzalvo
Jochen Haller
Ferdy Hanssen
Keith Harrison
Karl E. Hartel
Ludovic Henrio
Jaap-Henk Hoepman
Fabrice Huet
Günter Karjoth
Jeroen Keuning
Markus Kuhn
Law Yee Wei
Stefan Lucks
Eric Madelaine
Wenbo Mao
Antoni Martínez-Ballesté
Henk Muller
Heiko Oester
Francis Olivier
Béatrice Peirani
David Plaquin
Stéphanie Porte
Florence Quès
Michael Roe
Michael Rohs
Ludovic Rousseau
Torsten Schütze
Francesc Sebé
Nicolas Sendrier
Bernard Paul Serpette
Vasughi Sundramoorthy
Assia Tria
Michael Tunstall
Julien Vayssière
Lionel Victor
Harald Vogt
Peter Windirsch

Message from the Program Chair

Welcome to CARDIS, since 1994 the premier international research conference dedicated to smart cards and their applications. CARDIS '02, the joint USENIX/IFIP Iiternational Smart Card Research and Advanced Application Conference, brings together researchers and practitioners in the development and deployment of smart card systems and technologies. The Program Committee produced 201 reviews to select from the 45 manuscripts submitted. The 14 papers presented on November 21 and 22, 2002, represent the state of the art in smart card research and technologies.

I thank the Program Committee and the external reviewers for generously donating their time and effort in helping to select the program. I also thank the USENIX staff for their firm yet patient guidance in helping me as Program Chair. I also thank Pierre Paradinas and Vincent Cordonnier for their support and advice. Naturally, Pieter Hartel and Jean-Jacques Quisquater, the other two-thirds of the IFIP WG 8.8 leadership, were instrumental in setting the agenda, and they were always quick to lend a hand at a moment's notice. Finally, on behalf of the Program Committee, I thank the 97 co-authors who submitted papers to the conference, and especially the authors of accepted papers for their cooperation.

On behalf of IFIP WG 8.8 and the USENIX Association, I welcome you to San Jose. I encourage you to network with the world's leading smart card reseachers, but also to enjoy the beautiful California climate, never more beautiful than in late autumn.

Peter Honeyman
Program Chair

CARDIS '02: Fifth Smart Card Research and Advanced Application Conference

November 21–22, 2002

San Jose, California, USA

Implementing Group Signature Schemes With Smart Cards

Sébastien Canard and Marc Girault
France Telecom R&D
42 rue des Coutures - BP 6243
F-14066 Caen Cedex, France
{sebastien.canard,marc.girault}@francetelecom.com

Abstract

Group signature schemes allow a group member to sign messages on behalf of the group. Such signatures must be anonymous and unlinkable but, whenever needed, a designated group manager can reveal the identity of the signer. During the last decade group signatures have been playing an important role in cryptographic research; many solutions have been proposed and some of them are quite efficient, with constant size of signatures and keys ([1], [6], [7] and [15]). However, some problems still remain among which the large number of computations during the signature protocol and the difficulty to achieve coalition-resistance and to deal with member revocation. In this paper we investigate the use of a tamper-resistant device (typically a smart card) to efficiently solve those problems.

1 Introduction

In 1991, D. Chaum and E. van Heijst [8] introduced the concept of group signature schemes. A group signature scheme allows members to sign a document on behalf of the group in such a way that signatures remain anonymous and unlinkable for everybody but a group manager (GM), who can recover the identity of the signer whenever needed (the latter procedure is called "signature opening"). Numerous group signature schemes have been published and some of them are quite efficient ([1], [6], [7] and [15]). In more recent ones, signatures and public keys are constant-size and security is well established, allowing them to be used in various applications such as electronic cash ([15]), voting or bidding systems ([12]). However some problems still remain among which the high computation cost of the signature, the coalition-resistance and member revocation.
In this paper, we investigate a completely different approach for carrying out group signature schemes, namely the usage of a tamper-resistant device - typically a smart card. This allows a very low cost during the signature phase. In fact, the signer only has to compute two or three modular exponentiations (in contrast with roughly a dozen in the scheme from [1] for example). Moreover, the coalition-resistance problem is very easy to solve when using smart cards and more simple procedures can be used for member revocation.
The use of a smart card allows to prevent an (untrusted) member from cheating, by letting his (trusted) device both secretly store the signature keys and control their legitimate usage. Using smart cards allows to provide solutions for member revocation that are generic (i.e. work with any group signature scheme) and efficient, in that the signatures are short and constant-size, and the number of computations (for the signer and the verifier) is constant. Moreover the work during the revocation protocol is constant. Since smart cards are more and more used in real-life applications, our solutions can be implemented at a negligible extra-cost.
This paper is organized as follows. The following section provides background on group signature schemes and points remaining problems out. Section 3 presents our group signature scheme and shows that it is coalition-resistant. Section 4 presents various solutions for providing member revocation. Finally, we conclude in section 5.

2 Group Signature Schemes

This section presents the state of the art in the group signature area. It briefly introduces the security properties and then the related works.

2.1 Definition

Definition 1. *A group signature scheme is a signature scheme which satisfies the following properties:*
(i) Correctness: a signature produced by a group member is always valid.
(ii) Unforgeability: only group members are able to sign messages on behalf of the group.
(iii) Anonymity: given a valid group signature, it is infeasible for everyone but the group manager to identify the actual signer.
(iv) Unlinkability: deciding whether two different valid signatures were computed by the same group member is infeasible.
(v) Exculpability: neither a group member nor the group manager can sign on behalf of other group members.
(vi) Traceability: the group manager is always able to open a valid signature, i.e. to identify the actual signer.
(vii) Coalition-Resistance: a colluding subset of group members should not be able to generate a valid signature that the group manager cannot link to one of the colluding group members.

2.2 Related Works: Group Signature Schemes

Since the paper of Camenisch and Stadler [7], the same method has always been used to set a group signature scheme up. It is based on a difficult problem implying two or more values. Alice is a member of the group if and only if she knows a solution of this difficult problem.
If Alice wants to become a group member, she interacts with GM (who holds a secret key) in order to obtain in a blind manner her private key and her membership certificate. This latter value allows GM to establish the link between a signature and a group member.
During the signature protocol, Alice encrypts her membership certificate, then "proves" that she knows a solution of the difficult problem and that she has correctly encrypted her certificate. As a consequence, this protocol involves numerous modular exponentiations. Someone who wants to verify the signature only has to verify the whole proof, also known as a signature of knowledge. The group manager can open the signature by decrypting Alice's certificate.

Coalition-resistance has often be defeated ([7]) and was an unsolved problem until [1] and [6]. In these two articles, the authors propose new group signature schemes based on the strong RSA assumption ([3] and [9]) and prove that they are resistant to coalitions.

2.3 Related Works: Member Revocation

At any time a member can decide to leave the group. In this case, we can reasonably think that he will not try to cheat in the future, but it is far from sure. Furthermore if a member is revoked from the group against his will, it is very plausible that he will try to keep on signing even if he has not the right to anymore. In both cases, it is necessary to set up a mechanism which prevents this type of fraud.
The paper of E. Bresson and J. Stern [4] proposed the most intuitive solution which consists for the signer in proving that he is different from any revoked member. But this method obviously generates a signature whose size linearly increases according to the number of revoked members.
In a recent paper, Song [14] proposed two revocation methods that are relatively similar and provide constant-length signatures and a constant work for the group manager. But the work of the verifier is also linear in the number of revoked members. Moreover, the solution is not very practical since it deals with a group with a limited life-expectancy.
Ateniese, Song and Tsudik [2] proposed a modification of the Ateniese et al. scheme [1] to improve member revocation, which also provides a constant size of signature. But works during the revocation phase and the verification one are linear in the number of revoked members. Finally, the cost of the signature is very expensive and consequently it is an overall unpractical solution.
Very recently, Camenisch and Lysyanskaya [5] proposed the first practical method for member revocation. It is also based on the scheme of Ateniese et al. [1] and therefore is not really generic (i.e. cannot be easily applied to any other group signature

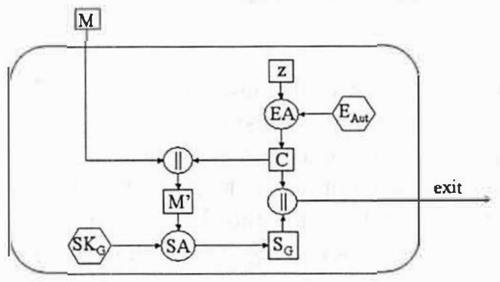
scheme). Moreover the signer has to make (possibly off-line) a number of modular exponentiations which is proportional to the number of modifications in the group (addition or deletion) until his last signature. Finally, this solution implies additional proofs of knowledge and, consequently, many other modular exponentiations.

3 Group Signature Schemes and Smart Cards

In this paper, we propose to build a group signature scheme relying on (typically) a smart card. It enables us to obtain straightforwardly the integrity of the (public or secret) data and of the program implemented in this tamper-resistant device. Moreover the confidentiality of keys and data is in the same way easily well-preserved. As a consequence, a solution simpler than previously proposed ones ([1] or [6]) can be introduced.

3.1 Shared Private Key and Smart Card

Our solution consists in using a smart card and a group-shared private key. First of all, we must choose an ordinary signature scheme (keys SK_G and PK_G) and a semantically secure cryptosystem (keys D_{Aut} and E_{Aut}), which is a cryptosystem where the ciphertext does not leak any partial information whatsoever about the plaintext that can be computed in expected polynomial time (and consequently, it is a probabilist cryptosystem). Then, the group manager computes keys in such a way that he can keep secret private ones (D_{Aut}) or distribute them (SK_G) to members without knowing them (for example, several group managers can share a discrete logarithm as the private key). He publishes public keys (PK_G and E_{Aut}).
If Alice wants to become a new group member, she firstly has to hold a smart card. Then, she has to obtain from the group manager an identifier z (which is unique and that identifies her) and the shared private key SK_G (which is common to all group members). Alice's smart card also has access to all parameters so as to use the cryptosystem (among which E_{Aut}) and the signature scheme defined above. The group manager has to keep in mind the link between the identifier (i.e. z) and the identity of the group member (i.e. Alice).
When Alice wants to sign a message as a group member (see Figure 1), she has to use her smart card. First, the identifier z is encrypted (algorithm EA) with the group manager's public key E_{Aut} (so that the group manager is the only one who can decrypt). Then the message M is concatenated with this encrypted value C and the whole is signed with the help of (algorithm SA and) the shared private key SK_G. As a consequence, only group members can sign a message and everybody is able to verify the signature with the associated public key PK_G.



M = Message
$||$ = Concatenation algorithm
z = Member's identifier
M' = Concatenation of M and C
EA = Encryption algorithm
SA = Signature algorithm
E_{Aut} = GM's encryption key
S_G = Signature of the message
C = Encryption of the identifier
SK_G = Group-shared signature private key

Figure 1: Shared Private Key and Smart Card

The verifier obtains the encrypted value C, the message M, and the signature S_G of the whole. He only has to verify the signature to be sure that the message is sent by a group member (because only group members possess the group-shared private key used to compute the signature). The group manager can open the signature by decrypting the identifier (with the key D_{Aut}).
It is important to note that the encryption scheme can either be symmetric or asymmetric. Nevertheless, it must be probabilist. On the contrary, it is necessary to use an (asymmetric) signature scheme for obvious reasons.
This approach makes possible a very fast signature, since there is only one encryption and one ordinary

signature to compute. Consequently, our solution is much better than previous ones in terms of speed and memory and in terms of genericity (any signature scheme can be employed).
Furthermore, it can be used in an on-line/off-line manner as follows : first of all, the card precomputes several encrypted values C in an off-line phase. Then, by using an on-line/off-line signature scheme SA, the card can precompute some values in an off-line phase, and later (in the on-line phase) produce group signatures very quickly, for example by doing a single multiplication if using the algorithm known as GPS ([10] and [13]).

3.2 Coalition-Resistance

The problem of coalition-resistance is easily solved when using tamper-resistant devices. In fact, it is impossible for two members to create a new card because they cannot access to protected data. In particular, they have no knowledge about the group-shared secret key SK_G (only their cards have). Moreover, producing a signature without knowing the private key violates the security assumption of the underlying signature scheme.

3.3 Security Arguments

Theorem 1. *Under the assumption that a smart card is tamper-resistant, the group signature scheme proposed in section 3.1 is secure.*

Proof. (sketch of)
We have to show that our scheme satisfies all the security properties that are listed in Definition 1.
(i) Correctness: by construction.
(ii) Unforgeability: only group members can have the private group-shared key in their smart card (due to their interaction with the group manager) and consequently can sign on behalf of the group.
(iii) Anonymity: everybody has the same private signature key and the identifier of the signer is encrypted. As a consequence, a verifier cannot identify the signer because each group member can potentially compute the same signature and he cannot learn anything from the encrypted value (see semantically secure cryptosystem).
(iv) Unlinkability: group members have a shared key and the cryptosystem is semantically secure. It is then infeasible to link two different signatures.
(v) Exculpability: this is due to the fact that the identifier of a signer is embedded in his group signature and that the smart card is tamper-resistant. Moreover, this property is ensured w.r.t the group manager since he doesn't know the group-shared key (see the first paragraph of section 3.1).
(vi) Traceability: the card always encrypts the identifier of the group member. As a consequence, the group manager can always decrypt it and then open the signature.
(vii) Coalition-Resistance: see the remark in section 3.2. □

4 Revocation in Group Signature Schemes

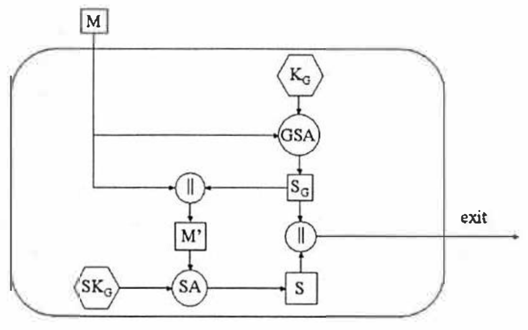
We suggest two approaches for dealing with member revocation. The first one is based on a group-shared private key and, as in section 3, relies on the confidentiality of this key (even w.r.t. the card-holder). The second one is based on "black lists" and relies on the integrity of the "black list" membership program executed by the card.

4.1 First Approach

4.1.1 General Principle.

Our approach consists in generating an additional signature computed with a group-shared private key SK_G. We denote by PK_G the associated public key. SK_G is communicated by the group manager to each non revoked member, by the means of a group key distribution scheme (for example [16]). As a consequence, the revocation problem is reduced to a group key distribution problem, for which solutions already exist. Moreover, it happens that, in our case, these solutions are easier to use.
When a new member wants to integrate the group, the group manager securely sends him, among other elements, the group-shared key SK_G. And when a member is revoked, the group manager sets up a mechanism of member revocation, which implies the renewal of the group-shared key. It is impossible for the revoked member to learn anything about the new shared key and consequently he cannot sign anymore. The group manager has to publish data in order to make possible for other members to get

the new key.
After that, if a member wants to sign a message M on behalf of the group (see Figure 2), he computes his group signature as usual (using [1], [6] or the solution described in section 3 for example) to obtain a couple (M, S_G) which he is going to sign by means of SK_G. The receiver can then verify the latter signature with PK_G and the value S_G as a signature of a group member.



M = Message
M' = Concatenation of M and S_G
K_G = Group (private/secret) key(s)
SK_G = Group-shared signature private key
GSA = Group signature algorithm
SA = Signature algorithm
S_G = M's group signature
S = Signature of the message
$||$ = Concatenation algorithm

Figure 2: First Approach - Signature Protocol

4.1.2 Group Key Distribution.

The most simple solution to manage group key distribution for our proposal is to share a secret key with each group member and to encrypt the new group-shared key with each secret key. Each valid member can decrypt one of the encrypted values to obtain the new group-shared key.
The identifier of the group member can be appended to each encrypted value. The group member only has to test if it is his own identifier and to decrypt the corresponding value if it is the case (see Figure 3).

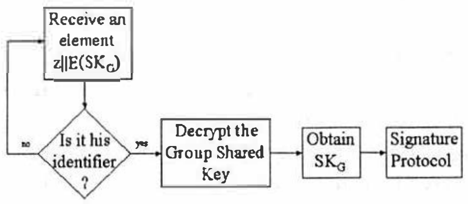


Figure 3: First Approach - Getting the Key

There are some other solutions in the literature that are more interesting than this simple one. For example, Wong et al. [16] propose a solution based on a tree, where each leaf corresponds to a group member and where each node corresponds to a secret key. Each group member shares with the group manager all keys that are in the path between their leaf and the root. As every member knows the key root, this latter is chosen as the group-shared key. Consequently, for a particular revocation phase, the GM only has a limited number of values to encrypt, instead of many in the naive method.

4.1.3 Security and Efficiency Considerations.

There is no way for the revoked member to learn anything about the new group-shared key. Then, the key contained in his smart card is no longer valid. As a consequence, the second signature will never be correct anymore. Finally, the group manager can efficiently and securely revoke group members.
The size of the signature is constant and the group signature is only increased by a single classical signature. Moreover, this method can be applied to any group signature scheme (including the one of section 3) and there is no extra work for the verifier (the cost is constant). The revocation protocol depends on the group key distribution scheme which is used. In particular, its cost will be at most linear in the number of group members.

4.1.4 Shared Private Key and Smart Card : Dynamic Case.

Section 3 presents a new group signature scheme based on a shared secret key and a smart card. Section 4.1 presents a solution to the problem of revocation that adds to the general group signature an ordinary signature that depends on a group-shared key. If one wants to apply this revocation method to this group signature, each signer will have a priori to compute two different signatures. But the two signatures can easily be merged into a single one, since they both use a group-shared secret key. This leads to a very attractive method which allows revocation while generating only one signature. More precisely, only one signature is necessary because it is possible to replace the (fixed) group-shared key of section 3 with a dynamic group-shared key, as explained in section 4.1. The group-shared key used in the group signature scheme only needs to be modified by the group manager after each revocation (see. section 4.1.2) and the rest is unchanged. Figure 1 shows the mechanism carried out by the smart card during the signature phase to which must be added the key updating phase illustrated in Figure 3.

4.2 Second Approach

4.2.1 General Principle.

Generally speaking, the simplest idea to deal with revocation problem is to maintain a revocation list (or a "black list"). The signer reveals a personal value and the verifier is then able to say, by matching the received value against each entry of the "black list", if the person is revoked or not. Unfortunately, in the context of group signatures, it is not possible to reveal a personal value since it would compromise the anonymity of the signer. Using a smart card allows to give a simple solution to this problem. Figure 4 shows the general principle of this approach.

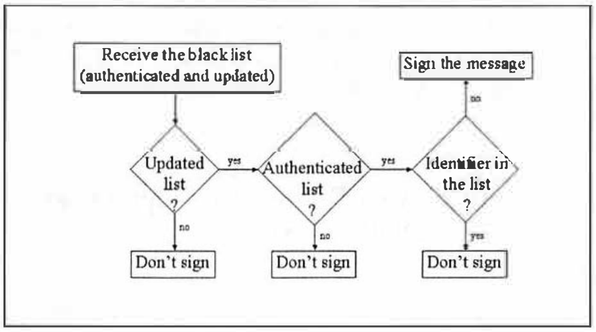


Figure 4: Second Approach - General Principle

In a few words, each member owning a personal value (an identifier), the smart card will get the revocation list from the group manager database (or any database where the "black list" stands, e.g. the verifier device) and will check if one value of the list and its personal value match. If the card reaches the end of the list, it will accept to sign as a group member; and if its personal value lies in the list, then the card will refuse to sign and make itself out of order.

4.2.2 First Solution.

Description. The first solution is straightforward and Figure 5 shows its principle. It consists in having the whole "black list" signed by the GM. Assuming that the underlying hash function of the signature scheme is iterative (most of them are so), it is possible for the smart card to verify the signature of a large message without needing to keep the entire message in his memory.

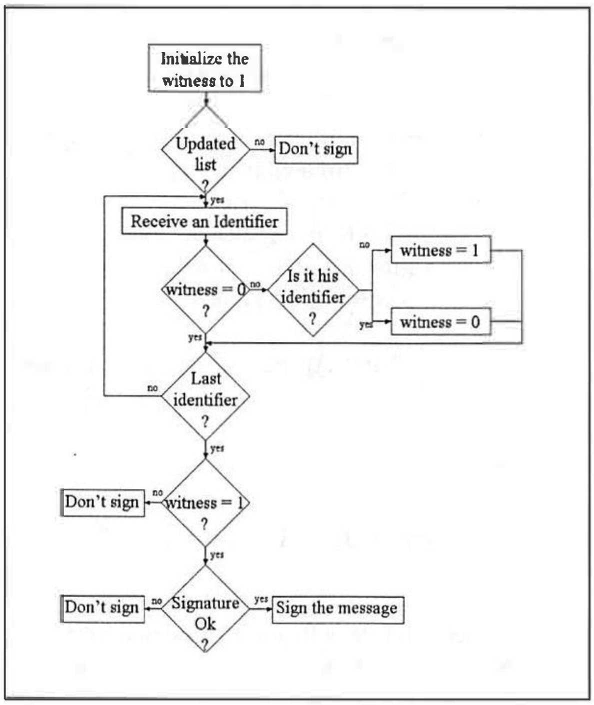


Figure 5: Second Approach - First Solution

Note that it is possible to use this method in a context of "white list" (that is a list which contains the identifiers of all members). In this case the card accepts to sign only if its identifier is in the list. It can be useful if the group has few members but a lot of revocations. We do not treat this case in this paper as it is an easy adaptation of the "black list" case.

Security. The mechanism is secure under the assumption that the card is tamper-resistant. In fact, an attacker who wants to add some more values in the revocation list cannot do it because he cannot falsify the group manager signature. Then, it is impossible to substitute a value for another one because the signature would then be incorrect. Moreover removing a value from the revocation list would generate a card error because the final test on the signature verification would be wrong. Finally, replaying indefinitely the same revocation list would imply the rejection of the signature by the verifier because he could compare the date of the updating by GM (D_{GM}) with the date of the last signature by the smart card (D_C). In fact, if D_C is different from D_{GM} he can think that the signer has intended to cheat. For example the revocation list can be updated every day. Another solution is the use of an on-line verification (even if it is an "extreme" case). We can then conclude that the previous mechanism is secure under the assumption that the card is secure.

Efficiency Considerations. This is a generic solution with a constant size of signature. In fact, the size of the signature is the same as that of the underlying signature scheme. From a computational point of view, there is a number of equality tests that is proportional to the number of revoked members, which can be considered as negligible, and the verification of only one signature. Another advantage of this solution is that the verifier does not have any extra computation to do. His work is no greater than that of the verifier in the underlying signature scheme. The work during the revocation phase is also constant. The group manager only has to add a value in the revocation list and to modify the resulting signature.

4.2.3 Second Solution.

Description. The second solution is also straightforward (see Figure 6). It consists in sending to the card all elements of the "black list" one by one, each of them signed by the group manager. It is yet necessary to add a *revocation number* (a sequence number: number 1 corresponds to the first revoked member, etc.) to prevent some attacks (for example addition or substitution of some identifiers). In addition, GM signs the date of his updating of the "black list" D_{GM} and the number of revoked members.

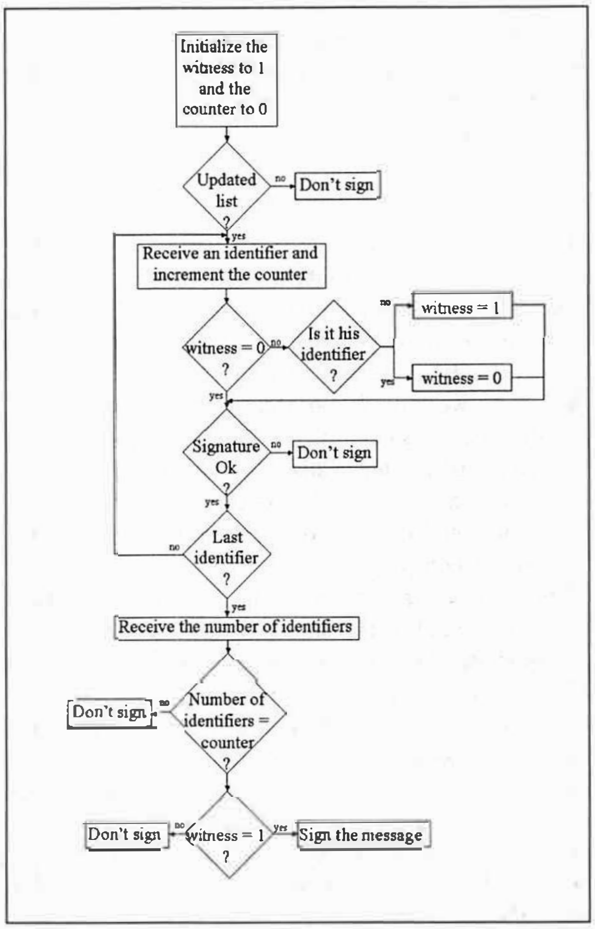


Figure 6: Second Approach - Second Solution

Security. The mechanism is secure under the assumption that the card is tamper-resistant. In fact an attacker cannot add some more values in the revocation list because he cannot afterwards compute the related signature. He cannot substitute a value for another one because the corresponding signature would then be incorrect. Removing a value from the revocation list would generate a card error because the final test on the signed number of revoked members would be wrong. Finally, as for the first solution (see section 4.2.2), there is no way to replay indefinitely the same list.

Efficiency Considerations. This is a generic solution with a constant size of signature. Once again, the size of the signature is the same as that of the underlying signature scheme. However, the signer has to check the validity of GM signatures for each revoked member which makes his work linear in the number of revoked members. The work of the group manager is constant-size since he only has to add a new value and to compute two signatures at each revocation. The verifier also has a constant-size work. Note that this method can also be used in a context of "white list".

An Improvement. At first glance, this solution seems to be less attractive than the first one. Indeed, the number of signatures to be verified is large if there are many revoked members. But a modification can be done so as to improve it.
Actually, we can argue that nobody can see nor modify the data exchanged between the smart card and the card reader. This is a plausible assumption if we consider that each member of the group has got a personal card reader that is always linked to his proper computer.
Therefore we can improve the solution by putting on a new value in the smart card memory that corresponds to the number of values that the card has already verified in the group manager database. Indeed, the card does not need to test twice the same values. Consequently, it can inform the card reader of the number of values it has already tested and as a consequence the card reader will only send to the card the new values since the last signature of that card (plus the signature of the updating date and of the number of revoked members). As a result, the card will only have a limited number of GM signatures to verify before producing a signature.

4.2.4 Third Solution.

A variant of the first solution consists in replacing the "black list" by a much shorter digest, so that the verification step becomes in average much faster. If the output of this step is "no", then we are sure that the member is not revoked and the card accepts to sign. However, if the output is "yes" then we cannot definitely conclude and the whole "black list" should be requested for a complete verification. We now briefly describe in the following subsection a possible way of achieving a compression of this kind.

An Example of Representation. The mechanism named "Superimposed coding" [11] allows to store a set of data of variable size into a bit-string of fixed size. It is then possible, with a simple test, to estimate the probability that an element is in the set of data (which depends on the size of the result bit-string and on the number of data). This probability is equal to 0 if the output of the test is "no". More precisely, the result is an m-bit string named B. We note $B = b_{m-1} \dots b_1 b_0$ where each $b_i \in \{0,1\}$. Initially, B is set to $00 \dots 0$. We have then k elements $y_1, \dots, y_k$ of various size and we note the set of data $Y = \{y_1, \dots, y_k\}$. Moreover, let us define q hash functions $h_1, \dots, h_q$ where each $h_i : \{0,1\}^* \longrightarrow \{0,1\}^c$ with $m = 2^c$.
For $j = 1..k$ we compute $h_1(y_j), \dots, h_q(y_j)$ and for every $l = 1..q$ we put to 1 the bit b_i where $i = h_l(y_j)$.
To know if the element y_R is in the set of data $Y = \{y_1, \dots, y_k\}$, we compute for every $l = 1..q$ $Y_l = h_l(y_R)$ and if there is an element $l_0 \in \{1, \dots, q\}$ such as $b_{Y_{l_0}} = 0$ then $y_R \notin Y$. If not, then $y_R \in Y$ with an error probability of about $\left(1 - e^{\frac{-kq}{m}}\right)^q$.

Description. The group manager uses the "Superimposed coding" to transform the set of all personal keys of each revoked member into the m-bit string B. Then he signs the latter value. A smart card is going to receive this signed bit-string, then treats it so as to verify the signature and to learn if its holder is revoked or not.
According to the size of the group and more particularly to the number of revoked members, the size of the result bit-string and the number of packets will vary in order to obtain good trade-offs (negligible error probability and m of reasonable size). For example, for $q = 8$ and $k = 10000$ (i.e. at most 10000 revoked members), the error probability is 2.3×10^{-5} for a result bit-string of size 2^{18} (i.e. 32 Kbytes).

Efficiency Considerations. This method is very interesting as the size of the signature and the number of computations remains constant and the resulting scheme is completely generic. Moreover, the size of verification work is constant. During the revocation protocol, computations are very simple and relatively independent from the number of revoked members, as the revocation manager only has to modify the resulting chain and has to compute the new linked signature. The only drawback is the probability of mistake, but since it can be made negligible, this third solution seems to be the more attractive one.

5 Conclusion

We have introduced a new way of designing group signature schemes by using a tamper-resistant device (as a smart card). First we showed how to build a (coalition-resistant) group signature scheme starting from any (ordinary) signature scheme and any (semantically secure) encryption scheme. Such group signatures can be computed very efficiently (typically only one or two exponentiation(s)) and are constant-size. Then we addressed the member revocation problem and solved it by using two approaches: in the first one, the group signature is completed with a signature involving a group-shared key which is renewed at each revocation; in the second one, the card checks it does not lie in a "black list" before computing a group signature. As a result, smart cards allow to design group signature schemes which are simple, generic, efficient and secure at the same time.

Acknowledgments

We are very grateful to David Arditti and Jacques Traoré for numerous discussions and comments. We would also like to thank the anonymous referees for their useful remarks.

References

[1] G. Ateniese, J. Camenisch, M. Joye, G. Tsudik. A Practical and Provably Secure Coalition-Resistant Group Signature Scheme. In L. Bellare, editor, Advances in Cryptology-Crypto'2000, volume 1880 of LNCS, pages 255-270. Springer-Verlag, 2000.

[2] G. Ateniese, D. Song and G. Tsudik. Quasi-Efficient Revocation of Group Signatures. In Financial Cryptography 2002, Southampton, Bermuda, March 11-14, 2002.

[3] N. Barić, B. Pfitzmann. Collision-Free Accumulators and Fail-Stop Signature Schemes Without Trees. In W. Fumy editor, Advances in Cryptology-Eurocrypt'97, volume 1233 of LNCS, pages 480-484. Springer-Verlag, 1997.

[4] E. Bresson, J. Stern. Efficient Revocation in Group Signatures. In K. Kim, editor, Public Key Cryptography-PKC2001, volume 1992 of LNCS, pages 190-206. Springer-Verlag, 2001.

[5] J. Camenisch, A. Lysyanskaya. Efficient Revocation of Anonymous Group Membership Certificates and Anonymous Credentials. Crypto'2002, to appear.

[6] J. Camenisch, M. Michels. A Group Signature Scheme based on an RSA-variant. Technical Report RS-98-27, BRICS, Dept. of Comp. Sci., University of Arhus, preliminary version in Advances in Cryptology-EUROCRYPT'98, volume 1514 of LNCS.

[7] J. Camenisch, M. Stadler. Efficient Group Signature Schemes for Large Groups. In B. Kaliski, editor, Advances in Cryptology-CRYPTO'97, volume 1296 of LNCS, pages 410-424. Springer-Verlag, 1997.

[8] D. Chaum, E. van Heyst. Group Signatures. In D. W. Davies, editor, Advances in Cryptology-Eurocrypt'91, volume 547 of LNCS, pages 257-265. Springer-Verlag, 1991.

[9] E. Fujisaki, T. Okamoto. Statistical Zero-Knowledge Protocols Solution to Identification and Signature Problems. In A.M. Odlyzko, editor, Advances in Cryptology-Crypto'97, volume 1294 of LNCS, pages 16-30. Springer-Verlag, 1997.

[10] M. Girault. Self-Certified Public Keys. In D.W. Davies, editor, Advances in Cryptology-Eurocrypt'91, volume 547 of LNCS, pages 490-497. Springer-Verlag, 1991.

[11] D. E. Knuth. The Art of Computer Programming, Volume 3 / Sorting and Searching. Addisson-Wesley Publishing Compagny. pages 559-563. 1973.

[12] K.Q. Nguyen, J. Traoré. An Online Public Auction Protocol Protecting Bidder Privacy. Information Security and Privacy, 5th Australasian Conference-ACISP 2000, pages 427-442. Springer-Verlag, 2000.

[13] G. Poupard, J. Stern. A Practical and Provably Secure Design for "on the Fly" Authentication and Signature Generation. In K. Nyberg, editor, Advances in Cryptology-Eurocrypt'98, volume 1403 of LNCS, pages 422-436. Springer-Verlag, 1998.

[14] D. Song. Practical Forward Secure Group Signature Schemes. ACM on Computer and Communications Security. 2001.

[15] J. Traoré. Group Signatures and Their Relevance to Privacy-Protecting Off-Line Electronic Cash Systems. In J. Pieprzyk, R. Safavi-Naini, J. Seberry, editors, Information Security and Privacy, 4th Australasian Conference-ACISP'99, volume 1587 of LNCS, pages 228-243. Springer-Verlag, 1999.

[16] C. K. Wong, M. G. Gouda, S. S. Lam. Secure Group Communications Using Key Graph. Technical Report TR-97-23, July 28, 1997, revised version in IEEE/ACM Transactions on Networking, Feb 2000.

Smart Cards in Interaction: Towards Trustworthy Digital Signatures

Roger Kilian-Kehr Joachim Posegga
SAP AG Corporate Research, CEC Karlsruhe
Vincenz-Priessnitz-Str. 1, D-76131 Karlsruhe, Germany
{roger.kilian-kehr, joachim.posegga}@sap.com

Abstract

We present approaches to raise the security level in the process of electronic signature creation by shifting as many tasks as possible involved in digitally signing data into a tamper-resistant and trustworthy smart card. We describe the fundamental technical principles our approach is based on, illustrate resulting design options, and compare the security of our approach with traditional electronic signature scenarios.

Keywords: *electronic signatures, smart cards.*

1 Introduction

The cryptographic underpinnings of electronic signatures such as mathematical one-way functions or public key cryptography are well understood, and practically secure algorithms and key lengths are widely established. From this perspective, electronically signing documents is a straightforward undertaking.

The actual procedure for digitally signing a document or a transaction, however, is a complex scenario in practice which involves numerous issues beyond cryptography: Since a person who wants to create a digital signature will usually not carry out the relevant computation by herself, she needs to delegate this to some application running on a platform (device) that can perform such computations. The security level of the overall signature creation process therefore depends on the security of several other, non-cryptographic factors, e.g. the environment where the document/data presentation takes place, the security of the communication channel to a user, or the security properties of the environment where the cryptographic computations are carried out.

Consider a scenario where a user signs a document displayed in a Web browser on a PC; at best, this involves a smart card, where the signing key is stored, and the cryptographic algorithm to encrypt the document hash (and other relevant data) is executed within the card. An attacker who wants to trick the user into signing a fake document would likely not attack the smart card, but the environment within it is used (i.e.: the OS, the driver of the smart card reader, the signing application, the Web browser, etc.).

The "added value" of a smart card is such a scenario is, largely, that it makes it hard to compromise the cryptographic key, but the card contributes little to the actual trustworthiness of an individual digital signature: The card is used as a tamperproof device that executes a fixed computational function, i.e., it reads a data block, encrypts it, and it returns the result. The card itself, however, does not interact directly with the user (card holder), but through a mediator like a PC or a mobile phone. But these devices are usually a lot less secure that a typical smart card.

This problem is, in theory, easy to solve: Raise the security level and require a closed, trustworthy system for applying electronic signatures. Unfortunately, this solution is extremely hard to roll out in practice, both because it is expensive and since dedicated hardware, which would be required, simply does not fit into today's computing world.

We propose to take another direction, and build upon execution platforms that are provided by today's smart cards, in particular SIMs and USIMs used for GSM and UMTS. Such cards offer functionality beyond the "hardwired", secure token that smart cards are mostly figured: Besides holding a secret key and performing cryptographic algorithms, GSM SIMs and UMTS USIMs include application platforms (e.g. [7, 15]), that allow programs that run inside these smart cards to use services of its host. A mobile phone hosting such a SIM provides I/O and networking capabilities to the SIM over standardized protocols [6, 3, 4, 5, 1, 2]. As a result, applications running inside a SIM can actively initiate and control user interaction, communicate over the network, etc.

Our paper discusses the various options for enhancing the security of the process of signing a document by involving a secure execution platform in such a smart card; essentially we investigate the following question underlying such an approach:

> *How much in terms of security can be obtained, if as much functionality as possible is shifted from untrusted components into a trustworthy platform available in a tamper-resistant device?*

Overall, our research provides means for increasing the trustworthiness of digital signature by imposing less assumptions on the integrity of a card terminal that classic approaches do.

Paper Outline

The main research contribution of our paper is given within Sect. 2. After introducing some notational conventions with standard digital signatures in Sect. 2.1, we investigate basic, on-card hash computation in Sect. 2.2. Section 2.3 extends this by involving a trusted third party. A third approach integrating the identity of the document's originator into the signature protocol is presented in Sect. 2.4. Although all the approaches are vulnerable to so-called "conspiracy attacks" they represent significant improvements in the overall security of an electronic signature creation process.

Based on the results of the previous approaches Sect. 2.5 proposes to digitally sign user interactions triggered by scripts that run inside smart cards to enable the comfortable, application-driven creation of electronic signatures on mobile devices.

Section 3 compares our work to related approaches, and we finally wrap-up our work in Sect. 4.

2 Smarter Signing with Smart Cards

This section explores several options for implementing the process of digitally signing documents by taking advantage of secure application platforms in smart cards: We discuss the security benefits of moving more and more of the required computation into the secure environment of a card.

As the starting point, consider the "traditional" procedure, where smart cards are used as crypto tokens holding a secret key and providing an implementation of cryptographic algorithms.

2.1 Basic Electronic Signature Protocol

The most important roles in scenarios for electronic signature creation are the signer S owning a public key pair (S_S, P_S), the document to be signed D, the signature creation application A, a document viewer V interacting with the signer, a smart card C, and the originator O of the document D. The basic protocol is as follows:

$$\begin{array}{lrl} (1) & O \rightarrow A: & \{D\} \\ (2) & A \rightarrow V, S: & \{D\} \\ (3) & S, V \rightarrow A: & accept/reject \\ (4) & A \rightarrow C: & \{h(D)\} \\ (5) & C \rightarrow A \rightarrow O: & \{sig_{S_S}(h(D))\} \end{array}$$

Here, (1) denotes the document transfer from the originator to the signature creation application, (2) the document presentation, (3) the signer's interaction/choice, (4) the hash computation, and (5) the signing process.

The above procedure can be improved w.r.t. security when moving some of these individual steps partially into the secure environment of a smart card. First we consider on-card hash computation.

2.2 Electronic Signatures with On-Card Hash Computation

The computation of the hash function is certainly a possible target for an attacker who wants to manipulate a signing procedure; but performing the hash computation inside a trusted device such as a smart card itself is not a panacea: it is important how the document presentation *and* hash computation is done in the overall signature protocol. Consider for example the following case:

$$\begin{array}{lrl} (1) & A \rightarrow C: & \{D\} \\ (2) & C \rightarrow A: & \{sig_{S_S}(h(D))\} \end{array}$$

In this case, (1) denotes the document transfer to the smart card and (2) the document signing process. From a security point of view an intruder I who is in control of A can easily exchange document D with another document D' which is subsequently sent to the card, hashed, and finally signed. Hence, compared with the basic protocol, no additional benefit can be gained from moving a hash computation into a card in a straightforward way.

On-Card Hash Computation Protocol

Assuming a scenario in which the signature creation application A is located in a security module C, and the viewer in a (less trustworthy) terminal a possible protocol is as follows:

$$\begin{array}{lrl} (1) & O \rightarrow A: & \{D\} \\ (2) & A \rightarrow V, S: & \{D\} \\ (3) & S, V \rightarrow A: & accept/reject \\ (4) & C, A \rightarrow O: & \{sig_{S_S}(h(D))\} \end{array}$$

Here, (1) denotes the document transfer to the application being hosted by the smart card, (2) the document presentation, (3) the user's choice, and (4) the hash and signature computation in the card.

Assuming end-to-end secure communication between O and *A/C*, an intruder is not able to control the hash computation anymore. Only the document presentation and the user's *accept/response* could be manipulated, although the intruder controlling V cannot gain anything from such manipulation, except by mounting the following attack.

A Conspiracy Attack on On-Card Hash Computation

- The intruder I and the originator O cooperate.
- O sends the document D', i.e., the document which the attackers want to be signed by S.
- Upon invocation of V, I presents a fake document D, which S might accept for signing.
- In the card, D' is signed and sent back to O.

Hence, an attack is still possible, if the intruder subverting V and the originator O of the document directly cooperate. Although this attack is of general importance, practically, it means that it is not sufficient anymore to attack the user's terminal only, but also to manage to actively send a faked document which the user subsequently signs.

As a consequence, shifting the hash computation in the above manner to a tamper-resistant device seems to give a substantial improvement in the overall security of the signature creation process.

2.3 Electronic Signatures Assisted by a Trusted Third Party

On-card hash computation is often not feasible, e.g. due to the limited bandwith one can use when communicating with a smart card. The process of computing hashes can, however, also be delegated to a trusted third party T as the following protocol outlines. A URL url_D is used to denote some resource where D can be fetched from. The trusted third party T then computes D's hash on behalf of A and signs it. A just forwards the URL to the document viewer V and the further protocol steps are the same as in the on-card hash computation protocol (cf. Sect. 2.2).

$$\begin{array}{lrl}
(1) & O \rightarrow A: & \{url_D\} \\
(2) & A \rightarrow T: & \{url_D\} \\
(3) & T \rightarrow A: & \{sig_T(h(D))\} \\
(4) & A \rightarrow V, S: & \{url_D\} \\
(5) & S, V \rightarrow A: & accept/reject \\
(6) & A \rightarrow C: & \{sig_T(h(D))\} \\
(7) & C \rightarrow A \rightarrow O: & \{sig_{S_S}(h(D))\}
\end{array}$$

In this protocol, (1) denotes the transmission of the URL under which the document to be signed is located to the application, (2) passing the URL to the TTP, (3) TTP fetches document and computes the hash, (4) represents the document presentation to the user, (5) the user's choice, (6) pass-through of the TTP's signature to the card and verification the the signature, and (7) the final signature computation by the smart card.

Similar to the on-card hash computation protocol, it is vulnerable to a conspiracy attack as described above.

2.4 Electronic Signatures with Recipient Addressing

Looking at the traditional signature creation protocol it becomes obvious that authenticity of a document sender is not of particular concern. In electronic business processes, however, signatures are often used to provide the technical means for contracts between two parties. Although the identities of the contract partners are usually somehow denoted in the document D, this is by no means cryptographically protected.

To improve the signature process further, we include the cryptographic identity of the document originator into the signature process. In particular we propose the following protocol which is based on the on-card hash computation protocol (cf. Sect. 2.2) and the public key pair (S_O, P_O) of the originator O denoted by id_O:

$$\begin{array}{lrl}
(1) & O \rightarrow A: & \{D, sig_{S_O}(D)\} \\
(2) & A \rightarrow V, S: & \{D, id_O\} \\
(3) & S, V \rightarrow A: & accept/reject \\
(4) & C, A \rightarrow O: & \{sig_{S_S}(h(D), sig_{S_O}(D))\}
\end{array}$$

Here, (1) denotes the document and signature transmission, (2) the presentation of the document and the identity of the originator, (3) the user's choice, and (4) the final hash and signature computation.

This protocol now achieves that an electronic signature is created over both – the cryptographic hash of the document and the identity of the recipient or originator of the signature.

To assess the advantages of this approach consider that in a traditional signature attack scenario an intruder could "hijack" the signing process of an arbitrary document D_O with its intended recipient O to infiltrate another document D' to be signed. The intruder I could then claim that the user has signed this document which is likely of advantage to the intruder. In the above protocol, however, the intruder I is not able to obtain a signature $sig_{S_S}(h(D'), sig_{S_I}(D'))$ since the signature $sig_{S_I}(D')$ cannot be generated. At best $sig_{S_S}(h(D'), sig_{S_O}(D'))$ could be obtained, but leading to a contradiction between the information available in D' denoting I as the recipient and the envelope signature sig_{S_O}. Therefore, we argue that linking the document and the recipient in the signature gives advantages to standard electronic signature creation.

Basically, the same conspiracy attack presented in the on-card hash computation in Section 2.2 can be mounted in the recipient addressing scheme. Again, if originator O and intruder I cooperate, the user is not able to distinguish that signature creation occurs with a document that she does not intend to sign.

2.5 Electronic Signatures on Interactions

We have so far considered electronic signatures on standard clients, e.g. desktop PCs. One of the most problematic issues with electronic signatures on mobile devices is the fact that such signatures are computed over complex documents. In particular this means that according to current signature laws, e.g. those in Germany, the document must be presented to the user who then either accepts or rejects the subsequent signature creation. Hence, a document to be signed must be presented as a whole in a suitably rendered fashion, which is often difficult on small, mobile devices. The problem of encoding and subsequently displaying a document in a reproducible and standardized way has been extensively discussed by Scheibelhofer [13]. In his approach he uses XML style sheets defining mappings to a possibly certified rendering engine.

To tackle this presentation problem, we consider not only the presentation of a document but also the way the document is created. We argue that a document is often the result of some kind of interaction between a service provider, e.g. who offers goods, and a client who selects goods to buy. Finally, after all selections are made, a document containing the complete list of goods is presented and signed accordingly.

If such an interaction "document" is encoded as an executable script, the execution of the script is *deterministic* as long as all *non-deterministic* input which is received from "outside" the script such as user input, random number generator, persistent variables, etc. is recorded. A "document" over which the signature is computed is then comprised of

(a) the executed script,

(b) the persistent state used during the computation,

(c) all user input,

(d) all messages received from other communication channels,

(e) the current time and progress of execution,

(f) some platform characteristics such as version numbers, serial numbers, etc.

The signature can be easily verified by executing the script in a simulated environment using the recorded and signed input values. Thus, a signed document in this sense is not intended to be human-readable, but rather meant to record and log the interaction that happened between a service provider and a user.

A Smart Card Platform for Mobile Code

More concretely, we propose to use a secure platform for the execution of (remote) code in a smart card which functions as follows:

- The smart card implements an interpreter for mobile code written in a domain-specific language optimally supporting the intended application domain.
- A client such as a service provider sends messages containing so-called *scripts* written in the domain-specific language the card-resident interpreter understands.
- The card's runtime platform executes the script, handles user interaction, and sends back the responses to the client.
- The platform implements key management facilities in order to provide end-to-end security between the client and the smart card.

Such a platform must be *secure* in the sense that neither the mobile code nor the user is able to harm the platform's integrity. Furthermore, the platform gives certain guarantees to both – code and user – that the scripts are executed as intended and no information leakage or secret storage manipulation can occur by malicious code or an external attacker.

Thus, the platform acts as a *trusted computing base* running in a tamper-resistant device protecting the user from the code and vice versa.

```
 1  script {
 2
 3     provider "bidbiz.com";
 4     name     "bidbiz auction client";
 5     id       "20011223/24357";
 6     options  signed-interaction;
 7
 8     implementation {
 9        playtone;
10        push("News from bidbiz.com:\nBid in auction #3576 (Antique watch): EUR 63.");
11        display;
12
13        push(mark);
14        push("Place new bid?");
15        push("New bid...");
16        push("Cancel");
17        select;                  ←— User selects option: (int,'1')
18
19        push( 2 );
20        eq?;
21        if (true) goto end;
22
23  enter:
24        push("Enter new bid (>EUR 63):");
25        input;                   ←— User inputs new bid amount: (string,'70')
26        dup();
27        push(63);
28        le?;
29        if (true) goto end:
30        playtone;
31        push("Please enter a bid greater than EUR 63.");
32        display;
33        goto enter;
34
35  end:  sign-interaction;
36        response;
37        exit;
38     }
39  }
```

Figure 1. Mobile auction client with interaction signatures

Example: Mobile Auctions

For illustration purposes we provide an example illustrating our approach in the domain of mobile auctions (Fig. 1). This example is based on the one presented in [8], however, it has been extended to support the creation of signatures on user interaction.

The given example illustrates the use of the stack-based domain-specific language we use to write our scripts without going into full detail. An in-depth description of the language can be found in [10].

A script starts with header information about the name of the script and its provider (lines 3–5). Line 6 denotes that the script's execution should be implicitly signed by the interpreter. The `implementation` part (line 7) contains the actual program.

Lines 9–12 demonstrate how to display an initial message about the latest news of the online auction. Lines 14–18 show how the arguments for a user selection (primitive `select`) are pushed onto the stack marked by the initial marker set in line 14. After the selection has been performed the arguments including the marker are removed from the stack and the number of the selected item is available on the stack.

Lines 20–22 check, whether the subscriber selected item no. 2 (i.e. "Cancel") in which case a jump to the label '`end`' is performed. Otherwise an input dialog is opened in lines 25–26 and the input from the subscriber is returned on the topmost stack position and duplicated in line 27. Then the entered amount is checked in lines 28–30, whether its is greater than 64. Otherwise a text is displayed in lines 31–33 and execution resumes to the input dialogue (label '`enter`').

Finally, in line 36, the whole recorded execution of the script is signed and a signature object containing all the relevant information about the script's execution including the signature is pushed onto the stack. The signature object is then sent back to the originator in line 37 and execution terminates.

During execution the runtime environment collects the non-deterministic input from the various sources into a log $L = \{i_1, \ldots, i_n\}$ of inputs i_j. In the above example execution thus yields

$$L = \{(\texttt{int}, `1'), (\texttt{string}, `70')\},$$

i.e., for each input we record the type information and the data. The overall interactive log of an execution of script P with the identifier id_P is computed and returned together with additional platform information R to the original sender S as follows:

$$C \rightarrow S: \quad \{id_P, L, R, sig_{S_C}(hash(id_P, P, L, R))\}.$$

The receiver must be able to verify the authenticity of the signature by simulating the execution of the script according to the log L. Based on this simulation, the interaction of the script and the user can be replayed and the user's choices and inputs can be examined to take appropriate action.

The execution of the script should occur in a transactional context, i.e. if for some reason the execution is terminated, no signature is created.

Summary

Using signatures on runtime execution audits combined with recording user interactions as a means to implement non-repudiation is, to the best of our knowledge, a novel approach. We consider this approach particularly useful for our application domain for the following reasons:

- Due to the lack of user input and output facilities, performing all possible executions within the trust domain of the smart card is from a security point of view desirable.
- All interaction which leaves the trust boundary of the smart card is reduced to the bare minimum, i.e. to user interactions only.
- The approach is very flexible, since it offers scripts a full control over the way signatures are built, how encryption is performed, and how interaction takes place. As such it is able to offer applications means to implement security policies as needed.

Thus, our approach allows to take full advantage of the smart card as an open platform for running security-critical applications in the tamper-resistant context of the physical device. More precisely, it represents one instance of the on-card hash computation approach as presented in Sect. 2.2. Furthermore, it can be easily extended to also support the third-party assisted approach in Sect. 2.3 and the recipient addressing approach in Sect. 2.4 assuming available key management facilities as described in [10].

3 Related Work

The main contribution of our research is in increasing the trustworthiness of digital signatures by building as much as possible on the security properties of execution platforms in smart cards.

Alternatively, one can try to enhance the trustworthiness of devices; [11, 12] discuss portable end-user devices (POBs) and security modules and define a number of requirements to be made for such devices. They observe that trustworthy POBs do not exist and conclude that therefore the development of secure applications should concentrate on protocols and procedures. A related approach is, e.g. described in [9] that comprises two different devices, a PDA and a smart card, that together implement a security-sensitive application: the smart card does not perform its task without the PDA and the PDA cannot perform the task without the help of the smart card.

One of the key ideas of this paper is documenting user interaction involved with digital signatures; a suitable, lightweight scripting language suitable for on-the-fly download to smart cards has been proposed in [8, 10].

The actual runtime execution monitoring using an execution log has been investigated by Vigna as a means to protect the execution of mobile agents in hostile environments [14]. The sender of a mobile agent can use this signed execution log to verify whether the agent has been tampered with while executing on a remote agent platform.

4 Conclusion

Starting from the observation that the process of electronic signature creation is still vulnerable in many practical settings, we have proposed three protocol variants that aim at shifting functionality from untrusted components into a smart card.

The first option considers on-card hash computation combined with end-to-end secure transfer of the document to be signed from the originator to the smart card. Another approach uses a trusted third party to perform resource-intensive computation of the document's hash outside the card. The third approach is characterized by the integration of identity of the document's originator into the protocol eliminating further attacks. However, so-called "conspiracy attacks" in which an intruder and an originator cooperate are still, yet less easily, mountable.

Based on the new protocols a novel approach for the creation of electronic signatures based on a runtime execution platform for smart cards has been presented. This approach is able to include the different protocol options presented and is especially suited for use in mobile settings characterized by the limited device capabilities in terms of user input and output. We have illustrated our approach with an example in the domain of mobile auctions – an application that is ideally suited to be run on mobile phones.

Generally, we suggest that GSM SIMs and UMTS USIMs might be ideal candidates for hosting such a smart card platform. Our results demonstrate that improvements in the electronic signature creation process are feasible if the environment the creation takes place is suitably taken into consideration.

References

[1] 3rd Generation Partnership Project. *3GPP TS 31.112 V5.0.0 (2001-09) Technical Specification 3rd Generation Partnership Project; Technical Specification Group Terminals; USIM Application Toolkit (USAT) Interpreter Architecture Description (Release 5)*, Sept. 2001. Available at *http://www.3gpp.org*.

[2] 3rd Generation Partnership Project. *3GPP TS 31.113 V5.0.0 (2001-09) Technical Specification 3rd Generation Partnership Project; Technical Specification Group Terminals; USAT Interpreter Byte Codes (Release 5)*, Sept. 2001. Available at *http://www.3gpp.org*.

[3] European Telecommunication Standardization Institution (ETSI). *Digital cellular telecommunications system (Phase 2+); Security Mechanisms for the SIM application toolkit; Stage 2 (GSM 03.48 version 8.1.0 Release 99)*, Nov. 1999.

[4] European Telecommunication Standardization Institution (ETSI), Sophia Antipolis, France. *Digital cellular telecommunications system (Phase 2+, Release 98): Subscriber Identity Module Application Programming Interface (SIM API); Service description; Stage 2 (GSM 02.19 version 7.0.0 Release 1998)*, 2000.

[5] European Telecommunication Standardization Institution (ETSI), Sophia Antipolis, France. *Digital cellular telecommunications system (Phase 2+): Subscriber Identity Module Application Programming Interface (SIM API); SIM API for Java Card; Stage 2 (GSM 03.19 version 7.1.0, Release 1998)*, 2000.

[6] European Telecommunications Standard Institute. *Digital cellular telecommunications system (Phase 2+); Specification of the SIM Application Toolkit for the Subscriber Identity Module – Mobile Equipment (SIM–ME) interface (GSM 11.14)*, 1998.

[7] Java Card Technology. Specifications are available at *http://java.sun.com/products/javacard/*.

[8] R. Kehr and H. Mieves. SIMspeak: Towards an open and secure platform for GSM SIMs. In I. Attali and T. Jensen, editors, *Proceedings of International Conference on Research in Smart Cards: Smart Card Programming and Security, E-smart 2001, Cannes, France,*, volume 2140 of *Lecture Notes in Computer Science*. Springer-Verlag, September, 19–21 2001.

[9] R. Kehr, J. Posegga, and H. Vogt. PCA: Jini-based Personal Card Assistant. In R. Baumgart, editor, *Proceedings of Secure Networking – CQRE [Secure]'99, Düsseldorf, Germany*, volume 1740 of *Lecture Notes in Computer Science*, pages 64–75. Springer-Verlag, November 30 – December 2, 1999.

[10] R. Kilian-Kehr. *Mobile Security with Smartcards*. PhD thesis, Darmstadt University of Technology, May 2002.

[11] A. Pfitzmann, B. Pfitzmann, M. Schunter, and M. Waidner. Vertrauenswürdiger Entwurf portabler Benutzerendgeräte und Sicherheitsmodule. In *Proceedings of Verläßliche Informationssysteme VIS'95*, pages 329–350. Vieweg, 1995.

[12] A. Pfitzmann, B. Pfitzmann, M. Schunter, and M. Waidner. Mobile user devices and security modules: Design for trustworthiness. Technical Report RZ 2784 (#89262), IBM Research Division, Zurich, May 1996.

[13] K. Scheibelhofer. What you see is what you sign. In *Proceedings of IFIP conference on Communications and Multimedia Security (CMS '2001), Darmstadt, Germany*, May 21–22, 2001.

[14] G. Vigna. Cryptographic traces for mobile agents. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*. Springer-Verlag, June 1998.

[15] Windows for Smartcards. *www.microsoft.com/smartcard/*.

Security analysis of smartcard to card reader communications for biometric cardholder authentication

Luciano Rila and Chris J. Mitchell
Information Security Group
Royal Holloway, University of London
Surrey TW20 0EX, UK.
luciano.rila@rhul.ac.uk and c.mitchell@rhul.ac.uk

Abstract

The use of biometrics, and fingerprint recognition in particular, for cardholder authentication in smartcard systems is growing in popularity. In such a biometrics-based cardholder authentication system, sensitive data may be transferred between the smartcard and the card reader. In this paper we identify and classify possible threats to the communications link between card and card reader during cardholder authentication. We also analyse the impact of these threats. We consider five different architectures and use the threat analysis to indicate the relative security of the various possible architectures.

1 Introduction

1.1 Biometrics and smartcards

Biometrics has been widely recognised as a powerful tool for problems requiring personal identification. Most automated identity authentication systems in use today rely on either the possession of a token (magnetic card, USB token) or the knowledge of a secret (password, PIN) to establish the identity of an individual. The main problem with these traditional approaches to identity authentication is that tokens or PIN/passwords can be lost, stolen, forgotten, misplaced, guessed, or willingly given to an unauthorised person. Biometric authentication, on the other hand, is based on physiological or behavioural characteristics of the individual, such as fingerprints, and therefore does not suffer from the disadvantages of the traditional methods.

In parallel, smartcards have steadily become more popular. Their increasing storage capacity and processing capabilities have enabled their deployment in a widening range of applications, varying from support for PKI to decentralised systems requiring off-line transactions [1, 2, 3]. Generally any application using smartcards requires a method for cardholder authentication, and biometrics-based authentication has emerged as an appropriate technology.

Combining the security of biometrics and the computing power of a smartcard is a very elegant solution to cardholder authentication. On the one hand biometrics can provide the level of security required by applications using smartcards. On the other hand, smartcards enable the biometrics technology by offering a secure and portable way of storing the biometrics template, which would otherwise need to be stored in a central database. Fingerprint recognition appears particularly appropriate for use in biometric systems using smartcards.

A smartcard system is composed of two main physical units: the smartcard itself and the card reader. In biometrics-based cardholder authentication, transmission of sensitive data between the smartcard and the card reader may occur depending on how the biometric system is distributed between these two units. In this paper, we consider the security issues associated with the communications link between the smartcard and the card reader during the biometrics-based cardholder authentication process.

Before we set out the objectives of this paper in detail, it is important to clarify the biometrics-based cardholder authentication process.

1.2 General model for biometric authentication

According to [4], a general biometric system is composed of the following logical modules:

1. Data collection subsystem;
2. Signal processing subsystem;
3. Matching subsystem;
4. Storage subsystem;
5. Decision subsystem;
6. Transmission subsystem.

The data collection subsystem contains the input device or sensor that captures the biometric information from the user. It is the link between the physical domain and the logical domain. The signal processing subsystem receives the raw biometric data from the data collection subsystem and extracts the distinguishing features from the raw data, transforming it into the form required for matching. The matching subsystem receives the processed data from the signal processing subsystem and compares it with the biometric template retrieved from the storage subsystem. The matching subsystem measures the similarity of the submitted biometric sample with an enrolled reference template. Each comparison yields a score, which is a numeric value indicating how closely the submitted sample and the reference template match. The decision subsystem receives the score from the matching subsystem and, using a confidence value based on security risks and risk policy, interprets the result of the score, thus reaching an authentication decision. The transmission subsystem provides the system the ability to exchange information between all other subsystems. Figure 1 shows a block diagram for the general biometric authentication model.

Note that these are logical modules, and therefore some systems may integrate several of these components into one physical unit.

1.3 Scope and purpose

In this paper we focus on the security issues associated with the communications link between the smartcard and the card reader during fingerprint-based cardholder authentication. PIN-based cardholder authentication has been well researched and understood, giving rise to a variety of industry standards, such as [5, 6, 7]. Encryption is typically used to provide security for PINs during transmission, either from the keypad to the card (for local cardholder authentication) or from the keypad to a remote server (for remote authentication of the cardholder).

However, for the purposes of our analysis, we do not make any assumptions about encryption or other cryptographic protection of the card/card reader communications link. This is because, whereas PINs are very short, biometric samples, e.g. fingerprint images, are rather large, and the limited computational and storage capabilities of the card may severely limit the possibilities for such protection.

Given our focus on card/reader communications, and the objective of assessing the best level of integration of the biometric technology, we make certain other simplifying assumptions. We assume that the smartcard is a tamper-proof device and any transmission between biometric system modules taking place within the card is therefore secure. We do not discuss the impact of using fake biometrics, such as plastic fingers, to fool the system, although it was shown in [8] that this is a possible attack with the current technology. We feel that this issue concerns fingerprint-based biometric technology in a wider sense and is therefore beyond the scope of our discussion.

In previous related work [9], a number of weaknesses in the biometric system model have been identified, and countermeasures suggested. However, in that analysis no assumptions as to the actual architecture of the system are made, and the analysis is rather general in nature. By contrast, the main purpose of this paper is to understand what security gains can be made from the various possible levels of integration of the biometric system on the smartcard.

Depending on how the logical modules of the biometric system are distributed between the smartcard and the card reader, different threats may arise. We consider five scenarios for the biometric system and, for each scenario, we identify and classify possible threats to the communications link and assess the impact of these threats. In all scenarios we assume that the smartcard stores the template for the

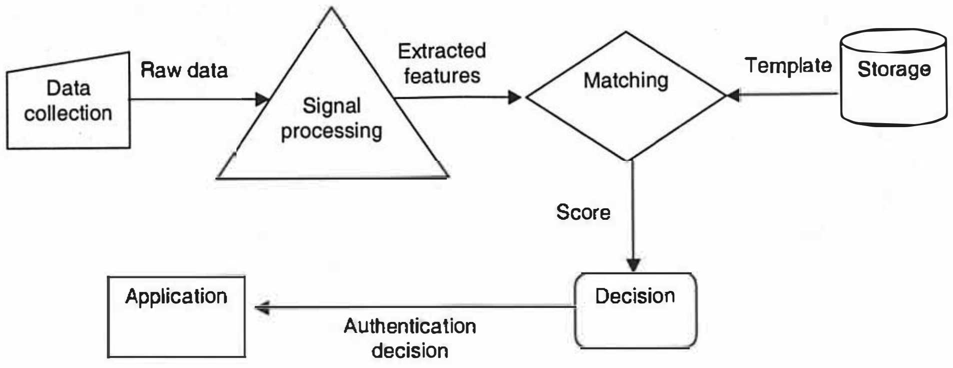


Figure 1: General model for biometric authentication.

cardholder fingerprint. We also assume throughout that fingerprint recognition is used as a method of cardholder authentication to the smartcard.

In Section 2 we describe the five biometric system architectures considered in this paper. In Section 3, we discuss the sources of communications link threats and then identify and classify the possible threats. In Section 4, we assess the impact of the threats identified in the previous section. Finally, we present our conclusions in Section 5.

2 Possible biometric system scenarios

Five different scenarios are considered, and the relative risks associated with each scenario are analysed. The scenarios cover various possibilities for the distribution of the modules of the biometric system between the smartcard and the card reader. Note that in all cases we assume that the fingerprint template is stored in the smartcard.

The scenarios are as follows:

S1. The fingerprint sensor is built into the card reader. The user template is transferred from card to reader. The reader takes the image provided by its built-in fingerprint sensor, performs the feature extraction, and also matches the features to the template provided by the card. The reader then informs the card whether or not authentication has been successful.

S2. The fingerprint sensor is built into the card. The fingerprint image and user template are transferred from card to reader. The reader performs feature extraction and matching of features to the template. The reader then informs the card whether or not authentication has been successful.

S3. The fingerprint sensor is built into the card reader. The reader takes the image provided by the built-in fingerprint sensor and performs the feature extraction. The extracted features are sent to the card, which then performs the matching process and reaches the authentication decision.

S4. The fingerprint sensor is built into the card. The fingerprint image is transferred from card to reader. The reader performs feature extraction only, and transfers the extracted features back to the card. The card then performs the matching process.

S5. All fingerprint processing takes place on the card.

Figure 2 shows the first four scenarios and their corresponding data flow during biometric cardholder authentication. Table 1 below defines all the scenarios in terms of the location of the various biometric modules.

3 Security threats

The focus of this paper is on the communications link between smartcard and card reader, and hence we only consider threats that relate, directly or indirectly, to this link. The main threats to this link can be divided into threats to the up-link (i.e. smartcard to reader) and down-link (i.e. reader to smartcard). The threats also vary depending on the scenario.

Note that, before identifying the threats to the up and down links, we briefly consider the possible source of these threats. Also, as well as identifying threats to the up-link and down-link, we briefly consider threats to the card reader itself. This is because the threats to the card reader indirectly relate to communications link protection (see below).

3.1 Sources of communications link threats

There would appear to be three main ways in which an attacker could intercept and/or manipulate data being transferred between card and card reader.

- The card reader (and/or smartcard) may emit electromagnetic signals which are data dependent, and which can be intercepted using an antenna located close to the reader. Such an approach would only enable passive (interception) rather than active (manipulation/replacement) attacks. The seriousness of this threat depends on the design of smartcard and card reader.
- A special interception device could be inserted into the read slot of the card reader, and the device would then be located between any inserted smartcard and the card reader. By this means, and without any modifications to the card reader, both passive and active attacks may be realised. The seriousness of such a threat will depend on a variety of factors including the design of the card reader and the environment in which the reader itself is located. Observe that, given that the primary threat would appear to arise from an attacker equipped with a lost, stolen or borrowed card, the seriousness of this threat will relate to whether or not use of the card reader is supervised by trusted personnel (who might detect the use of additional devices).
- The card reader could be modified. At the simplest level this could mean the insertion of a 'bug' designed to monitor and perhaps modify data communications. (See also Section 3.4 below).

We do not discuss the magnitude of these threats further here, since all three threats are very much implementation-dependent and therefore any further analysis would be highly speculative. However, it is clear that, wherever possible, card readers should be designed to minimise these threats, particularly if sensitive information is transferred between smartcard and reader without cryptographic protection.

3.2 Up-link threats

The main up-link threats are as follows:

U1. (**S1** and **S2** only). Interception (leading to loss of confidentiality) of the user fingerprint template.

U2. (**S1** and **S2** only). Manipulation (or replacement) of the user fingerprint template.

U3. (**S2** and **S4** only). Interception (leading to loss of confidentiality) of the fingerprint image.

U4. (**S2** and **S4** only). Manipulation (or replacement) of the fingerprint image.

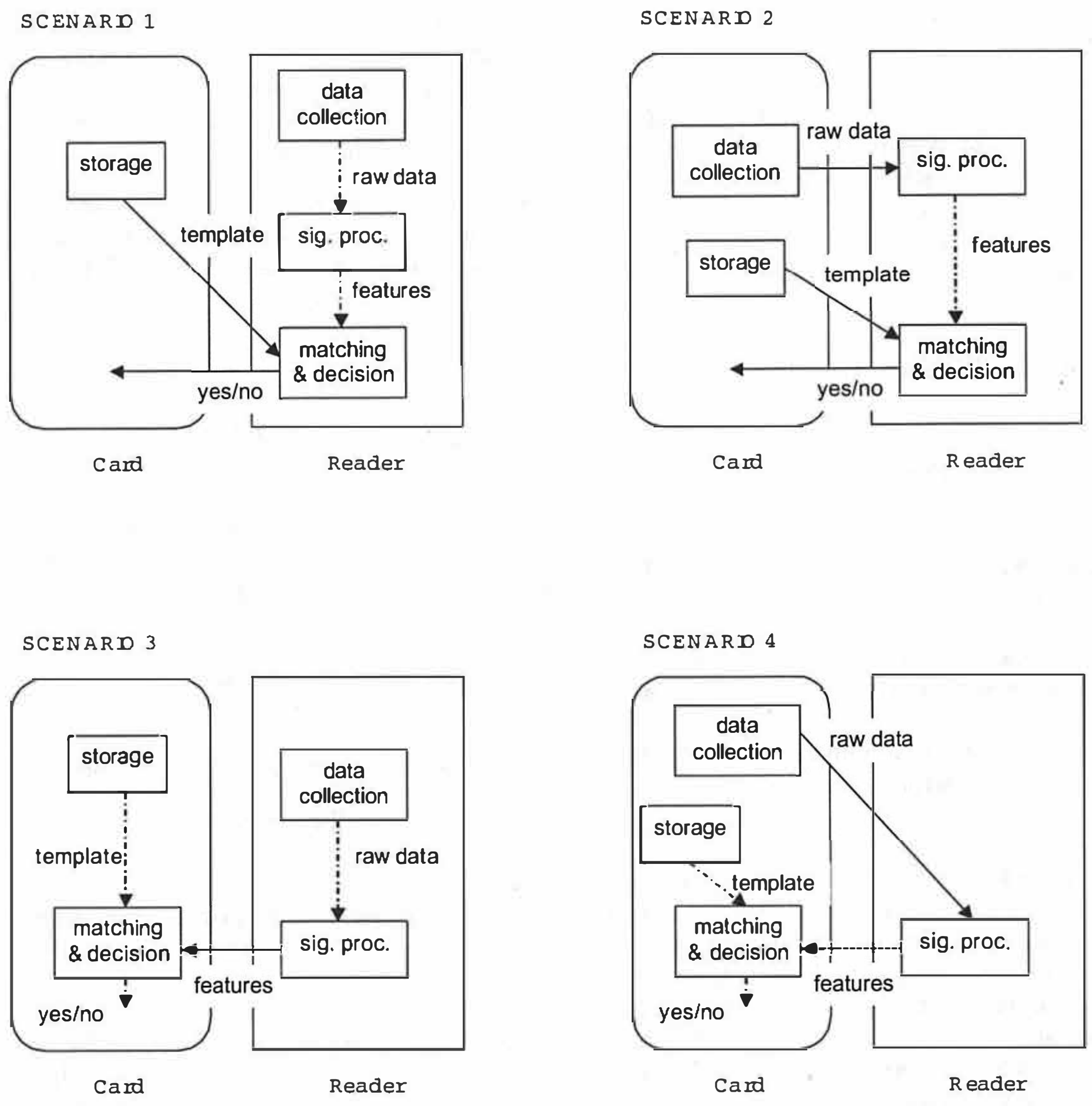


Figure 2: Four different scenarios and their corresponding data flow during cardholder authentication.

	System modules in smartcard	System modules in card reader
Scenario 1	Template storage	Data collection Signal processing Matching and Decision
Scenario 2	Template storage Data collection	Signal processing Matching and Decision
Scenario 3	Template storage Matching and Decision	Data collection Signal processing
Scenario 4	Template storage Data collection Matching and Decision	Signal processing
Scenario 5	All modules	No modules

Table 1: Five different biometric system scenarios in increasing order of integration of the biometric modules.

Threats **U1** and **U3** could be addressed by encrypting the communications path, although the effectiveness of such a measure would depend on the physical security of the card reader (since keys necessary to decrypt the transferred data would need to be available to the card reader). Addressing threats **U2** and **U4** would require the provision of data integrity and origin authentication services for the data transfer between card and reader (e.g. as provided by a Message Authentication Code (MAC) or a digital signature — see, for example, [2]).

3.3 Down-link threats

The main down-link threats are as follows:

D1. (**S1** and **S2** only). Modification of the authentication decision.

D2. (**S3** and **S4** only). Interception (leading to loss of confidentiality) of the fingerprint features.

D3. (**S3** and **S4** only). Manipulation (or replacement) of the fingerprint features.

Threat **D2** could be addressed by encrypting the communications path, although the effectiveness of such a measure would depend on the means used to protect the necessary key(s). Addressing threats **D1** and **D3** would require the provision of data integrity and origin authentication services for the data transfer between reader and card (e.g. as provided by a MAC or a digital signature).

3.4 Threats to card reader

Three main types of threat to the card reader can be identified. Although these threats are not directly relevant to smartcard/reader communications security, they do have indirect relevance (see below). The three main classes of threat are as follows.

- *Manipulation* of a genuine card reader. This includes the insertion of a 'bug' (as mentioned in Section 3.1), but also includes threats where the operation of the reader is modified, e.g. by changing stored software.
- *Replacement* of the card reader. This refers to the substitution of the genuine reader with a fraudulent replacement. (Whether or not this could be achieved without preventing correct operation of the system depends on both the card reader design and the design of the remainder of the system).
- *Theft* and/or *reverse engineering* of the card reader. Such a threat could be very serious if the reader contains secrets on which the system security depends.

4 Impact of security threats

We next consider the impact of the various threats identified in the previous section. We divide this discussion into the following sub-categories:

- threats arising from attempted use of a lost, stolen or borrowed card;
- threats to integrity of card transactions;
- threats to cardholder privacy.

4.1 Use of lost, stolen or borrowed cards

As a basis of this discussion we assume that the possessor of a misappropriated (lost, stolen or borrowed) card wishes to make use of this card, e.g. to perform some kind of transaction. In order to do so, he/she will need to find some way of 'fooling' the cardholder authentication process.

There are a variety of ways this could be achieved, as follows. Note that in each case we indicate which threat identified in Section 3 above is giving rise to the issue.

- Arising from **U2** (and hence applying to **S1** and **S2** only): replace the fingerprint template as sent on the up-link with a fingerprint template belonging to the possessor of the misappropriated card.

  For such an attack to be viable, the attacker will need to have a fingerprint template for his/her own fingerprint in the format used by the scheme. There are a number of possible ways in which this could be obtained.

- If the attacker has his/her own card, this could easily be obtained by monitoring the output from the attacker's own card.
- If the attacker knows the type of fingerprint reader in use (either built into the card reader (**S1**) or built into the card (**S2**)) and the method used to obtain the template, then the attacker could obtain a fingerprint reader of this type and use it, together with appropriate software, to compute a template.
- The attacker could use a misappropriated card to obtain a copy (or many copies) of a fingerprint image (from threat **U3** — i.e. **S2** only) for his/her own fingerprint. With knowledge of the method used to extract a template, together with appropriate software, the attacker could compute a template.

If **U2** is realisable, then this risk has to be classified as **high**, since, for many systems, protecting one cardholder against another fraudulent cardholder is a necessary requirement.

- Arising from **U4** (and hence applying to **S2** and **S4** only): replace the fingerprint image as sent on the up-link with a fingerprint image belonging to the legitimate cardholder.

  For such an attack to be viable, the attacker will need to have a fingerprint image for the genuine cardholder. There are a number of possible ways in which this could be obtained.

  - From threat **U3** (and hence applying to **S2** and **S4** only). Note that this would require threat **U3** to be realised before the time of misappropriation. This may not be easy to arrange.
  - If the attacker knows how the fingerprint reader in use operates, and has access to a fingerprint image of some kind for the genuine user (e.g. by taking an image from an object touched by the genuine cardholder) then it may be possible to transform this latter image into one conforming to the scheme in use.

  If **U4** and **U3** are realisable for the same card, then this risk has to be classified as **high**. Note that even if both threats are realisable, successfully taking advantage of both threats with respect to the same card may be much more difficult. If **U4** is realisable but not **U3**, then the risk is lower — say **medium** — depending on the details of the fingerprint imaging technology being used.

- Arising from **D1** (and hence applying to **S1** and **S2** only): change the authentication decision sent from reader to card from 'Reject' to 'Accept'.

  This is trivially easy to perform (given that threat **D1** is realised). If **D1** is realisable, then this risk has to be classified as **very high**.

- Arising from **D3** (and hence applying to **S3** and **S4** only): replace the fingerprint features sent on the down-link with features extracted from the cardholder's fingerprint.

  For such an attack to be viable, the attacker will need to have a copy of fingerprint features for a fingerprint of the genuine cardholder in the format used by the scheme. There are a number of possible ways in which this could be obtained.

  - From threat **D2** (and hence applying to **S3** and **S4** only). Note that this would require threat **D2** to be realised before the time of misappropriation. This may not be easy to arrange.
  - If the attacker knows how the fingerprint feature extraction method in use operates, and has access to a fingerprint image of some kind for the genuine user (e.g. by realising threat **U3** before the time of misappropriation, or by taking an image from an object touched by the genuine cardholder) then it may be possible to derive a workable set of features conforming to the scheme in use.

  If **D3** and **D2** are realisable *for the same card*, then this risk has to be classified as **high**. Note that even if both threats are realisable, successfully taking advantage of both threats with respect to the same card may be much more difficult. If **D3** is realisable but not **D2**, then the risk is lower — say **medium** — depending on the details of the fingerprint imaging and feature extraction technology being used. Note also that threat **D3** could be reduced if a secret feature extraction technique is used — for this to be effective the card readers in use would need to possess physical security features (see Section 3.4). Threat **D3** could also be reduced (if not eliminated) if it was possible for the card to verify that the fingerprint features provided by the card reader indeed belong to the image provided to the card reader.

Note that none of these threats apply to scenario **S5**, which is not prone to attack on the communications path since this path is not used for the cardholder authentication process.

We summarise the results of the above analysis in Table 2.

4.2 Card transaction integrity

Whilst there may be many risks to the integrity of card transactions, we restrict our attention here to the impact of threats to the card/reader communications link.

The only impact which results from the analysis described here is an indirect one. If any of the threats relevant to the particular scenario are realisable, then this may give a cardholder the ability to dispute transactions after they have occurred. That is, if a fraudulent cardholder knows of the existence of certain threats which would allow successful use of a lost, stolen or borrowed card, then the cardholder could, after completion of a genuine transaction, claim that the transaction had been performed by someone else using a lost, stolen or borrowed card.

4.3 Cardholder privacy threats

The other main area of impact of the threats identified in Section 3 is to the privacy of the cardholder. That is, the cardholder may have concerns relating to who has access to information relating to his or her fingerprint. Note that we are concerned here purely with privacy concerns, unrelated to any possible threat of fraud.

The following impacts arise from the identified threats.

- Arising from **U1** (and hence applying to **S1** and **S2** only): loss of confidentiality of user fingerprint template.
- Arising from **U3** (and hence applying to **S2** and **S3** only): loss of confidentiality of user fingerprint image.
- Arising from **D2** (and hence applying to **S3** and **S4** only): loss of confidentiality of user fingerprint features.

The three impacts are rather similar to one another, and all have an impact on user privacy. The choice of scenario (apart from the fact that **S5** is unaffected) has little bearing on the degree of the impact.

5 Conclusions

The main purpose of the analysis in this paper is to understand how to integrate biometric cardholder authentication with a smartcard in the most cost effective manner. In particular we have sought to understand what is to be gained from the various possible levels of integration of biometric system with the smartcard.

First and foremost it is clear that scenario **S5** is unaffected by the security of the communications path since in that scenario the card/reader communications path is not used (at least for the cardholder authentication process). Thus scenario **S5** is clearly the best in an absolute sense — however it is also likely to be the most costly to deploy. It is therefore interesting to understand how the other four scenarios compare, bearing in mind that, of these four, scenario **S1** is likely to be the cheapest option and scenario **S4** the most expensive, since they represent the lowest and the highest level of integration respectively.

For the other four scenarios, the cardholder privacy threat is very similar regardless of the scenario. The main issue would appear to be fraudulent use of misappropriated cards.

Of scenarios **S1**, **S2**, **S3** and **S4**, it would appear that scenarios **S1** and **S2** are very similar with respect to their vulnerability to attacks on the card/reader communications path. The degree to which scenarios **S3** and **S4** reduce the risk depends

Scenario	Degree of risk
S1	Very high (if **D1** realisable). High (if **U2** realisable).
S2	Very high (if **D1** realisable). High (if **U2** realisable). High (if **U3** and **U4** realisable for the same card). Medium (if **U4** realisable).
S3	High (if **D2** and **D3** realisable for the same card). Medium (if **D3** realisable).
S4	High (if **U3** and **U4** realisable for the same card). High (if **D2** and **D3** realisable for the same card). Medium (if **D3** realisable).
S5	None.

Table 2: Summary of impacts of misappropriated cards.

partly on technical issues relating to the format and use of fingerprint images and features, and also depending on how easy it would be to both steal a card and monitor its use prior to its theft.

S4 represents a higher level of integration of the biometric system with the smartcard than **S3**. However the integration of the fingerprint sensor with the smartcard in **S4** makes the system vulnerable to threats **U3** and **U4** in the uplink. From that point of view, **S4** would appear to be an architecture more open to attacks than **S3**. Note, however, that when the fingerprint sensor is built into the card reader, the system becomes vulnerable to threats to the card reader (see Section 3.4). As suggested in [9], a fake card reader could be used to record the biometric data of legitimate users in an attack similar to a false ATM attack, which may potentially be an attack more easily realisable and more damaging than threats **U3** and **U4**. Moreover, given that the sensor is a fragile piece of equipment, integrating the sensor with the card reader is not a viable solution for many applications since it makes the system vulnerable to vandalism.

The gain to be derived from integrating the fingerprint sensor with the smartcard is minimal if all fingerprint feature extraction and matching are done off the card. However, depending on the environment, significant gains can be achieved as long as the matching is performed on card, even when the feature extraction is performed off-card.

It is interesting to note that almost all the most serious threats arise from an assumed lack of integrity for the data link. If it is assumed that the card reader is a trusted device and has not been interfered with or replaced (see also Section 3.4), then guaranteeing the integrity of the link between the card reader and the card would effectively prevent all the threats, even in the absence of any confidentiality for data transferred.

Finally note that, given that the threats discussed mostly relate to use of misappropriated cards, the use of secure auditing and blacklisting measures within the application can help to minimise the impact of such threats.

6 Acknowledgments

The work described in this paper has been supported by the European Commission through the IST Programme under Contract IST-2000-25168 (Finger_Card). The information in this document is provided as is, and no guarantee or warranty is given or implied that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

The authors would like to thank their colleagues in the Finger_Card Project for their encouragement and advice.

References

[1] M. Hendry, *Smart Card Security and Applications.* Artech House, 1997.

[2] A. Menezes, P. van Oorschot, and S. Vanstone, *Handbook of Applied Cryptography.* CRC Press, 1997.

[3] W. Rankl and W. Effing, *Smart Card Handbook.* John Wiley & Sons, 2001.

[4] ISO/DIS 21352: 2001, Biometric information management and security, ISO/IEC JTC 1/SC 27 N2949.

[5] ANSI X9.8 — 1995, Banking — Personal identification number management and security — Part 1: PIN protection and principles.

[6] ISO 9564-1: 2002, Banking — Personal identification number (PIN) management and security — Part 1: Basic principles and requirements for online PIN handling in ATM and POS systems.

[7] EMV 2000, Integrated circuit card specification for payment systems, Book 2 — Security and key management. Version 4.0, 2000.

[8] T. van der Putte and J. Keuning, "Biometrical fingerprint recognition: don't get your fingers burned", in *Proc. 4th IFIP WG8.8 Working Conference on Smart Card Research and Advanced Applications (CARDIS 2000)*, Bristol (UK), 2000, J. Domingo-Ferrer, D. Chan, and A. Watson (editors), Kluwer Academic Publishers, pp. 273-288.

[9] G. Hachez, F. Koeune, and J.-J. Quisquater, "Biometrics, access control, smart cards: a not so simple combination", in *Proc. 4th IFIP WG8.8 Working Conference on Smart Card Research and Advanced Applications (CARDIS 2000)*, Bristol (UK), 2000, J. Domingo-Ferrer, D. Chan, and A. Watson (editors), Kluwer Academic Publishers, pp. 273-288.

Secure Method Invocation in JASON *

Richard Brinkman
Department of Computer Science, University of Twente
P.O. Box 217, 7500 AE Enschede, the Netherlands
brinkman@cs.utwente.nl

Jaap-Henk Hoepman
Department of Computer Science, University of Nijmegen
P.O. Box 9010, 6500 GL Nijmegen, the Netherlands
jhh@cs.kun.nl

Abstract

In this paper we describe the Secure Method Invocation (SMI) framework implemented for JASON, our Javacard As Secure Objects Networks platform. JASON realises the *secure object store* paradigm, that reconciles the card-as-storage-element and card-as-processing-element views. In this paradigm, smart cards are viewed as secure containers for objects, whose methods can be called straightforwardly and securely using SMI. JASON is currently being developed as a middleware layer that securely interconnects an arbitrary number of smart cards, terminals and back-office systems over the Internet.

1 Introduction

JavaCard[1] [Che00] technology makes it possible to develop software for a smart card using a high level language: JAVA. This technology is platform independent, it can handle multiple applications (each running securely within its own sandbox) on one smart card, post-issuance applications can be added to it and it is compatible with international standards like ISO7816 [ISO7816].

In fact, the JavaCard platform brought high level, Object Oriented Programming (OOP) to the smart card developer. Unfortunately, the OOP paradigm is only applied to the software within the smart card itself: invoking methods implemented by objects on the smart card still requires the developer to send commands to the smart card using Application Protocol Data Units (APDU's) [ISO7816], which have to be processed and transformed into method calls 'by hand'.

It would be much more natural to view an object stored on a JavaCard as a remote object, accessible through a remote method invocation mechanism. In fact, if we look at a smart card application at a higher level of abstraction, we basically see a large collection of interconnected objects. Some of these objects are stored in back offices, others in terminals or PC's and many more stored securely on millions of smart cards. This network is highly dynamic: smart cards are usually offline, and only connect to the network when they are inserted into a terminal (or when they connect to a terminal over a wireless interface in the case of contactless cards). Much more importantly, this network needs to be highly secure. Access to certain objects should be restricted, and the confidentiality and authenticity of the communication between the objects has to be guaranteed.

Hartel *et al.* [HJF95] pose that a smart card should be seen as a processing element

*Id: javacard-smi.tex,v 1.10 2002/09/23 06:04:03 hoepman Exp

[1] `http://java.sun.com/products/javacard`

rather than a storage element (as is traditionally done). In our opinion these views are not contradictory at all, but rather supplement each other nicely in the *secure object store* paradigm. In this paradigm, smart cards are viewed as secure containers for objects, whose methods can be called straightforwardly and securely using Secure Method Invocation (SMI). We are currently developing the Javacards As Secure Objects Network (JASON) platform as a middleware layer (on these smart cards, terminals, PC's and back office systems) to support this paradigm. By simplifying the communication with a smart card, and by providing extensive support to secure this communication, JASON aims to greatly simplify the development of smart card applications.

In this paper we will describe the JASON Secure Method Invocation (SMI) scheme. In this scheme, a JASON definition file (JDF) (resembling a JAVA interface with some additional keywords) is used to specify the access conditions on methods of an object. It also specifies how the parameters of a method call and the result should be protected when transmitted between caller and callee. The JDF is compiled into a *stub* (used by the caller to set up a connection with the object and to call its methods) and a *skeleton* (used by the callee to accept incoming method invocation requests and to handle the security requirements). The big advantage is that the smart card application developer only needs to specify the security requirements, but does not have to implement the security protocols himself. This is done automatically, given the requirements.

The remainder of this paper is organised as follows. We first present related research in the next section. Then, the main requirements for the JASON platform are presented in Sect. 2. The design (in terms of the application programmers view on JASON) is given in Sect. 3. Section 4 discusses the architecture and the way the JASON SMI is actually implemented, while Sect. 5 presents a small example of using JASON to implement a basic electronic purse. Finally, conclusions and issues for further research appear in Sect. 6

1.1 State of the art

Itoi *et al.* [IFH00] add security to the Internet infrastructure for smart cards developed by Guthery *et al.* [Gut00, GBPR00] and Rees *et al.* [RH00], adding the Simple Password Exponential Key Exchange (SPEKE) protocol and using the DNS as a location independent naming scheme for the smart cards involved. These aspects will be taken into account in the networking and naming part of the JASON platform.

Hagimont and Vandewalle [DH00] apply a different approach to enforcing access control on (remote) objects. Their JCCAP system uses capabilities to specify which methods of an object can be accessed by the owner of that capability. Capabilities are implemented through Java interfaces, and provide a limited view on the full interface of an associated object. This makes their system dynamic (in the sense that capabilities can be added and removed from the system independent of the actual implementation of the object, and that capabilities can be delegated between objects. On the other hand, they do not consider the general case of caller and callee residing on different systems separated by a network (as well as the terminal/card line interface). Moreover, the very important matter of protecting the data transfered with an actual method call is not considered in their work.

The latest JavaCard specification (2.2) includes a lightweight version of Sun's Remote Method Invocation (RMI) [Sun99]. It provides a mechanism for a client application running on the terminal to invoke a method on a remote object stored on the card just like an invocation within the same virtual machine. The parameters of a remote method should be primitive (byte, boolean, short, int) or a single-dimension array of a primitive type (byte[], boolean[], short[], int[]). Unlike standard Java RMI, object parameters (whether remote or not) are not allowed. The method result is of primitive type, a single-dimension array of primitive type, a remote interface object or void. All parameters and return values are transmitted by value, except for the remote object. The remote object is transmitted by reference. We have investigated several approaches to implementing our JASON Secure Method Invocation (SMI) system using RMI, but none are quite satisfactory. We discuss this in Sect. 4.4.

Keht *et al.* [KRV00] describe the JiniCard architecture, which allows seamless integration of smart card services in a spontaneous network environment. The approach taken is to keep all functionality required to interact with a certain smart card remotely on the network, and to download this functionality into the card reader based on the ATR (Answer To Reset) of the particular card inserted into it. They also discuss the service-as-object metaphor, but as far as security is concerned, they consider SSL sessions between card and terminal *objects* over which RMI calls are being sent. We, on the other hand, introduce a much finer security granularity at the method level.

There are also a number of related industry initiatives that deserve to be mentioned here.

The Global Platform Specification[2] (formerly VISA's Open Platform specification) is concerned with the secure and platform independent installing and deletion of applications on multi-application smart cards.

The Open Card Framework[3] (and the similarly motivated PC/SC Workgroup[4]) aims to allow software developers to build smart card-aware products without having to worry about platform, card terminal, or smart card-specific interfaces. It supplies an API for handling the communication between a PC application and a smart card reader. Since OCF is developed by the major smart card companies, it supports all kinds of smart cards and card readers. The application does not even have to know which smart card reader is being used during a communication session with a card. OCF does not specify the card side. The choice of a particular type of smart card is free and may change without changing the PC application.

[2]http://www.globalplatform.org
[3]http://www.opencard.org
[4]http://www.pcscworkgroup.com/

2 Platform requirements

With the JASON SMI system we want to achieve:

- Separation of concerns: specifying security requirements (in the interface definition of a JAVA applet using our keyword approach), and their actual implementation (provided *once* through the JASON SMI system).
- Generic secured access to objects and their methods, independent of their location and whether they are on a compute server or a smart card.
- Providing generic, interoperable, tools to secure method invocations, which can be shared among objects (decreasing the code size) and which can be verified once (increasing robustness and avoiding repeated verification of similar per-applet security measures).
- Decreasing the complexity of writing secure (smart card) applications.

3 Design

The JASON platform implements the secure object store paradigm using the following layers.

Network layer Implements the direct connection between clients, servers, terminals and smart cards, using the Internet Protocol. Between terminal and smart card IP packets are transferred as APDU's. In particular, a smart card (when inserted in a terminal) has an IP address, and the terminal acts as a gateway relaying all incoming IP packets to the appropriate smart card (it may contain more than one smart card) [GBPR00].

remote method invocation layer Serialises method parameters into bytestreams and vice versa, and executes the call on the remote method

secure method invocation layer Provides access control and data confidentiality and authenticity.

In this paper we will focus on the design of the secure method invocation layer, and describe it as seen from the application programmer's point of view. We will discuss the close interdependencies with the RMI layer. The SMI layer only requires of the underlying layers that it delivers messages at least to the intended recipient.

3.1 Main components

The Secure Method Invocation (SMI) layer allows a *caller* object to securely call a method implemented by a *callee* object. Both caller and callee are assumed to be stored and run in a protected environment (a *sandbox*) that disables access to all objects and data within the sandbox except through published interfaces.

The JASON SMI layer provides the following services:

- identification and authentication of caller and callee,
- role based access control at the method level, and
- confidentiality and authenticity of method parameters and results.

In future versions other services will be added like:

- logging
- transaction support
- non-repudiation

To call a method of an object, the caller first has to connect to the callee in a particular role. This establishes a *security context* between caller and callee, that (among others) contains the session keys used to protect the communication. Once connected, the caller can call all methods declared by the object accessible to this role. For JASON, roles are equivalent to keys. In other words, ownership of a particular key associated to a role, proves that an object can connect in that role.

To establish a connection, the caller needs a *stub* corresponding to the object to connect to. Similarly, the callee needs a *skeleton* that receives incoming connections, performs access control decisions and protects the method parameters and results. The role keys used to authenticate the caller to the callee are stored in a separate *keystore* object belonging to the same sandbox. This design is sketched in Fig. 1.

The stub and skeleton necessary to securely call the methods of an object are generated automatically from a so called JASON definition file. This file specifies the security requirements for the callee object. The contents and structure of this file are described next. Note that the issue of key management falls beyond the scope of this paper. We are currently investigating the proper tools to support key management within the JASON framework. As far as the JASON SMI platform is concerned, the keystore contains valid and proper keys.

3.2 The JASON definition file

The JASON SMI system has a strict separation between the card application and its security. An application developer has two tasks.

- Write a card object without bothering about security or APDU exchange, instead focusing on the information processing logic of the application.
- Write a JASON definition file describing the security requirements.

Therefore, the security requirements for an object are written in a separate JASON definition file that resembles the syntax of a JAVA interface description.

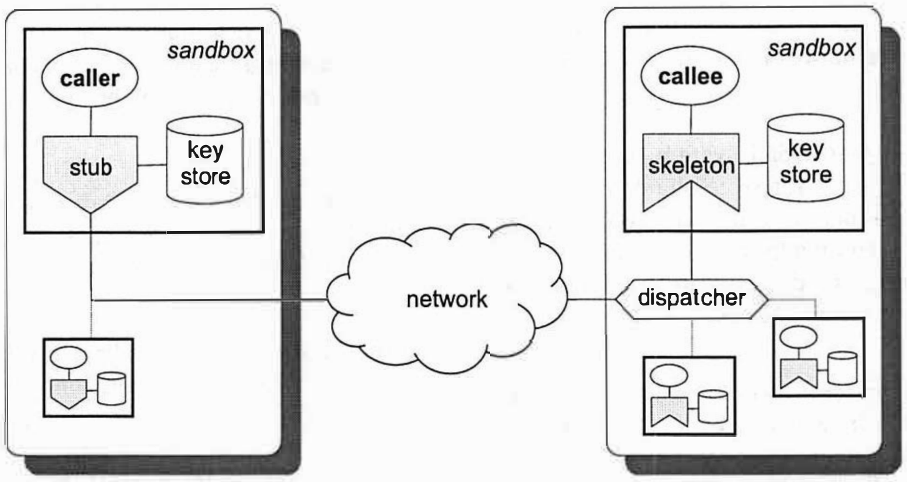


Figure 1: Caller and callee components.

```
package com.ebank;

public interface Purse
{
  roles BANK, MERCHANT, OWNER;

  accessible to ALL
  authentic short getBalance();

  accessible to BANK
  authentic short increaseBalance( confidential authentic short amount);

  accessible to MERCHANT
  authentic short decreaseBalance( authentic short amount);
}
```

Figure 2: JASON definition file for a simple purse.

A sample JASON definition file appears in Fig. 2 (describing the interface of a simple electronic purse application, that will be studied further in Sect. 5). The JASON precompiler will process the definition file and generates three files.

- A plain JAVA interface file. All keywords not known in JAVA are removed. This is the interface implemented by both the implementation of the callee object and the client stub.
- A client/caller stub, whose methods are called to execute the corresponding remote methods, and that performs authentication and marshalling (including protection) of data.
- A callee skeleton performing access control decisions, unmarshalling of parameters (verifying signatures and decrypting parameters where necessary) for incoming invocation requests, and executing the actual method.

In JAVA the keywords `private`, `protected` and `public` are used to limit access to methods and fields to certain classes. An object can only access it's own private members, protected members of it's superclasses or classes in the same package and all public members. These keywords work fine if used inside a single virtual machine. However, when using a distributed system a more fine grained solution is necessary.

In the JASON SMI system, access control is role based. Moreover, the communication between caller and callee has to be protected as well. To specify these requirements, the JAVA interface description is extended with the following keywords.

- `roles` ⟨*role-list*⟩, listing the different roles in which a caller can connect to this object. The roles in this list correspond to keys stored in the keystore.
- `accessible to` ⟨*role-list*⟩, specifying which roles can call the indicated method.
- `confidential` and/or `authentic`, specifying that a parameter or a method result should be confidential and/or authenticated.

Here a ⟨*role*⟩ is an identifier (usually in all caps because it is a constant), and a ⟨*role-list*⟩ is a comma-separated list of roles. Let us discuss the last three keywords in a little more detail.

`accessible to` ⟨*role-list*⟩ Access to a method can be limited by using the `accessible` keyword. Access is only to be granted if the caller can be identified (using the corresponding keys in the keystore) as a role in ⟨*role-list*⟩. The predefined role ALL indicates that access is allowed for all roles defined for this object (through the `roles` keyword). The predefined role ANYBODY specifies a role that can be assumed by anybody (i.e., a role whose identity is not verified). For security reasons only methods are accessible from off-the-card applications. Variables should be accessed through corresponding set and get methods.

`confidential` Parameters and return values can be specified as `confidential`, meaning that the data involved should be sent encrypted between caller and callee. This guarantees that nobody else can eavesdrop the value. In the negotiation phase (see below) a (symmetric) session key is exchanged and an encryption algorithm chosen.

`authentic` Parameters and return values can also be specified as `authentic`. This gives the following guarantees.

authenticity Only the caller can construct valid parameters[5], and only the callee can construct valid responses. The parameter received by the callee was sent by the caller, and the result received by the caller was sent by the callee. In particular, this gives the caller the guarantee that the intended side effects of the method call did in fact occur at the callee (like decreasing the balance of a purse).

integrity The parameter (or the result) received was not altered while in transit.

freshness The parameter received was passed by the caller for the *current* call of the method (and not for any previous call). The result received was sent by the callee for the current call of the method (giving the guarantee that the method was actually executed at this time, see above).

In practice this means that the data involved should be signed, and that a form of replay protection is added as well.

3.3 Using SMI

To call a method using the SMI framework, the caller has to perform the following two steps (see also Fig. 3 for an example connecting to the purse object whose interface was given previously).

- The first step is to connect to the callee and to establish a security context. The

[5]Strictly speaking, because a symmetric session key is used to protect the data, also the callee can construct valid parameters. Therefore non-repudiation cannot be guaranteed.

```
try {
  Purse purse = (Purse) SMINaming.connect("smi://smartcard/Purse",
      Purse.MERCHANT, purseKeyStore) ;
  try {
    purse.decreaseBalance(10);
    System.out.println("You have paid");
  }
  catch (UserException ue) {
      System.out.println("Transaction failed. You have not paid.");
  }
}
catch (RemoteException re) {
  System.out.println("Failed to connect to service.");
}
```

Figure 3: Caller connecting to a callee

caller passes the name and location of the desired service, the desired role in which to connect, and a reference to the key store to SMINaming.connect(). When successful, this returns a reference to the required stub.

- Subsequently, the methods of the remote object can be called securely as if they were local methods of the stub returned by the previous step.

If a connection is established, the stub also contains the current security context for that connection. Among other things, this security context contains a session key used to secure subsequent method invocations. Also, it contains further identification information on the callee object. This identity can be retrieved by the stub's `getSessionIdentifier()` method.

Note that even for a single call to a method, a connection has to be set up. This may be wasteful for certain applications where transaction speed is very important (e.g., public transport). We are investigating the possibility of calling a single method without connecting to the object first (in fact merging the connection and the calling into one step).

4 Architecture

In this section we describe how the JASON SMI platform is actually implemented, and how the security requirements are actually met using several cryptographic protocols. In particular we show how a secure connection is setup, how the ownership of roles is verified, and how the security context is established. Secondly, we show how a method is called securely using the information and session keys in the current security context. But first we will discuss the keys stored in the keystore in a little more detail.

4.1 On keys

The keys in the keystore correspond one-to-one to the roles declared in the JASON definition file. The keystore also contains keys for key-management. This is discussed in a forthcoming paper.

JASON supports the use of different types of keys in the keystore, depending on the security requirements of the application (or indeed individual objects on particular smart cards). Currently, the following types of keys are supported.

- RSA, with 512, 1024 and 2048 bit keys.
- DES and 3DES.

- AES, with 128, 192 and 256 bit KEYS.

Moreover, JASON supports *diversified* keys [AB96] where the key k_i stored by callee i (used by the callee to authenticate the caller or vice versa) is derived from the master key k_M stored by the caller. The key is derived using the formula

$$k_i = \{i\}_{k_M} ,$$

where $\{m\}_k$ denotes encryption of message m using key k (where the encryption method is defined by the type of the key). Note that in this case k_i performs the role of a public key (from which the corresponding private key cannot be derived), but with additional property that it proves to the *caller* the identity i of the callee.

Depending on the type of key stored in the keystore, the appropriate authentication protocol is run. Note that the caller keystore contains the keys necessary to prove its role (e.g., private keys), while the callee keystore contains the keys necessary to verify a role (e.g., public keys). If an entry in the caller keystore is `null` or invalid, the caller cannot assume the corresponding role. If an entry in the callee keystore is `null` or invalid, the role cannot be verified and all connections for that role will be refused.

Finally, the keystore contains, for each role key, information about the type of cipher that should be used to protect the session once the caller has been authenticated and accepted.

4.2 Connecting to an object

Connecting to an object exchanges and verifies the identity and role of the caller and the callee. Furthermore, a security context is established (containing a shared secret key) that is used to protect all calls to methods of the object. To connect to an object and establish a session the following steps are taken (assuming RSA style authentication).

- The caller sends a message containing
  - the role (as an index in the keystore) as which it wants to connect,
  - the type of key it will use to authenticate the role (RSA in this example),
  - a list of all ciphers it will accept to protect the session, and
  - a nonce.
- The callee looks up the role and the type of keys it can accept. If it can accept the suggested authentication method, it will select one of the ciphers to protect the session from the list it received (provided it supports it). It then sends the following message
  - the selected cipher to protect the session,
  - a random master secret encrypted with the public RSA key found for the role in the keystore, and
  - a nonce,
- The caller validates the proposed cipher, decrypts the master secret with its private key in the keystore.
- Both caller and callee generate the session key (using hashes) from the master secret and both the caller and the callee nonces.
- Caller and callee exchange further identifying information encrypted and MAC-ed using the session key, and record that in the security context.

Both caller and callee record the session key in the security context for this connection. Note that if a connection is established as `ANYBODY`, no verification of that role can be performed. In that case, the master secret must be exchanged using a Diffie-Helman type key exchange. Future method invocations are will be secured using this session key.

The session context also contains two counters, one to count the number of messages sent in this session, and one to count the number of messages received. Both are reset to 0 at the start of a session, and incremented for each message sent or received.

These numbers are used to protect against replay, as explained below.

4.3 Method invocation

Informally speaking, after session setup the stub and the skeleton are connected by a (secure) byte stream. The byte stream is routed by the communications layer to the correct skeleton. In fact, when a stub's method is invoked, it does the following:

- reconnect to the remote JVM containing the remote object,
- marshal(write and transmit) the parameters to the remote JVM,
- wait for the result of the method invocation,
- unmarshal (read) the return value or exception returned, and
- return the value to the caller.

The stub hides the serialisation of parameters and the network-level communication in order to present a simple invocation mechanism to the caller.

In the remote JVM, each remote object has a corresponding skeleton. The skeleton is responsible for dispatching the call to the actual remote object implementation. When a skeleton receives an incoming method invocation it does the following:

- unmarshal (read) the parameters for the remote method,
- invoke the method on the actual remote object implementation, and
- marshal (write and transmit) the result (return value) to the caller.

The byte stream sent from stub to skeleton contains the following elements.

- The name (or rather the index) of the method to call, together with a MAC computed using the session key and the current value of the sent messages counter. Even if RMI is used as the transport mechanism, this information is necessary to prevent remote method invocations being redirected to the wrong method.
- Each confidential parameter is encrypted.
- For each authentic parameter, a MAC computed using the session key and the current value of the sent messages counter is appended to the parameter.

For efficiency reasons parameters are shuffled so that the `confidential` and `authentic` parameters are placed in contiguous blocks within the byte stream (see Fig. 4). All confidential parameters are encrypted as a single block. Similarly, the MAC for all authentic parameters is computed in a single block, appending the sent messages counter only once.

The return stream from skeleton to stub to communicate results has the following structure.

- If the return type is confidential, the return value is encrypted with the session key.
- If the return type is authentic, the sent messages count is appended to the byte stream, and both the count and the value are used to compute a MAC with the session key. The result is appended to the byte stream.

4.4 Inter object communication

Because the caller and callee are physically separated by a network, the call to a remote method must be transferred to the remote object over the network before it can be executed there. The most natural approach would be to use Java's Remote Method Invocation mechanism to achieve this. At the caller side, the SMI stub first converts the

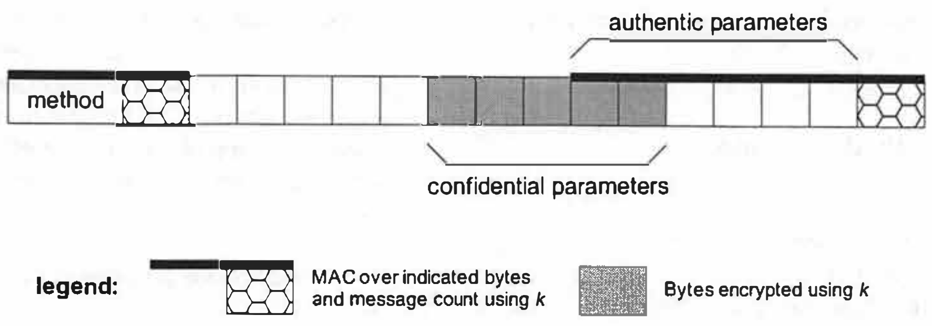


Figure 4: Byte stream structure from caller to callee.

parameters to a protected bytestream, as explained in Sect. 4.3. The RMI layer than transmits this bytestream to the callee, and invokes the corresponding method of the callee SMI skeleton. There, the access permissions are checked and the bytestream is unpacked before the original callee method is invoked.

However, this scenario is complicated by the fact that JavaCard (as of version 2.2) uses a different RMI system, if only because a JavaCard is not connected to a network directly, but instead communicates with the outside world through a terminal using an APDU stream. This would imply that the terminal has to convert an incoming RMI request to a JavaCard specific JC-RMI request (and similarly for the responses). This does not appear to be straightforward, because the RMI wire protocols are different. The only option is to create - for each skeleton on the callee smart card - a separate skeleton (and stub) for the terminal, that receives the incoming RMI request and simply calls the remote method on the smart card using JC-RMI. This means the terminal potentially needs access to a huge number of skeletons and stubs, simply to pass bytestreams verbatim!

Moreover, we note that RMI's support for marshalling and unmarshalling of method parameters and results becomes totally superfluous in this approach, because the SMI layer already converts the parameters to a bytestream in the first place.

To solve the first problem RMI and JC-RMI need to be brought more in line, such that their wire protocols become sufficiently compatible to allow translations between them using a generic translation mechanism running in the terminal. To solve the second problem, the RMI system should provide versatile hooks to allow the outgoing bytestream to be protected in the fine grained manner required by JASON. Or, SMI should be incorporated into the RMI layer.

5 Example

Fig. 2 in section 3.2 shows the security requirements of a simple purse application. It corresponds to the actual implementation given in Fig 5. Clearly the implementation is quite straightforward. Also, the strictness of the separation between implementation and its security is apparent. The implementation does not contain a single line of code concerning security. All the security is contained in the generated stub and skeleton. The skeleton calls the implementation and adds security to it. Note that each method is defined with the default JAVA visibility, to allow the skeleton to access them, but not giving access to subclasses outside the package.

6 Conclusions & Further Research

We are currently implementing the JASON SMI framework on a JavaCard 2.2 platform. The final implementation will be available under the GNU

```
package com.ebank;

class PurseImpl implements Purse {
  public static final byte OVERFLOW = (byte) 1;
  public static final byte UNDERFLOW = (byte) 2;
  private short balance = 0;
  private static final short MAX = (short) 500;

  short getBalance() {
    return balance;
  }

  short increaseBalance(short amount) throws UserException {
    if (balance + amount < MAX) {
      balance += amount;
      return amount;
    } else
      UserException.throwIt(OVERFLOW);
  }

  short decreaseBalance(short amount) {
    if (balance - amount > 0) {
      balance -= amount;
      return amount;
    } else
      UserException.throwIt(UNDERFLOW);
  }
}
```

Figure 5: Implementation of a simple purse.

General Public License (GPL) through `http://www.cs.kun.nl/~jhh/jason.html` within a few months.

We intend to extend JASON's SMI functionality with logging and auditing functions, as well as transaction (and rollback) support. Related to the logging and auditing issue, is the fact that the current implementation does not provide non-repudiation. The ramifications for implementing non-repudiation are the subject of further investigations. Also, one could argue that the `authentic` keyword is overloaded (in the sense that it gives too many guarantees, especially freshness, at the cost of a more complex and resource consuming protection mechanism). Using JASON to develop several real-world smart card applications will tell whether a more fine grained set of security specification keywords is required.

Finally, to make the JASON vision of a smart card application consisting of millions of distributed objects a reality, object broker functionality has to be added that is consistent with the high security requirements of typical smart card applications, and the highly dynamical nature of the smart card network.

7 Acknowledgements

We thank the anonymous referees for the valuable comments and suggestions, and especially for bringing to our attention that the JavaCard 2.2 standard contains support for RMI.

References

[AB96] ANDERSON, R. J., AND BEZUIDENHOUDT, S. J. On the reliability of electronic payment systems. *IEEE Trans. on Softw. Eng.* 22, 5 (1996), 294–301.

[Che00] CHEN, C. *Java Card (tm) for Smart Cards: Architecture and Programmer's Guide.* The Java Series. Addison-Wesley, 2000.

[DH00] D. HAGIMONT, J.-J. V. Jccap: capability-based access control for java card. In *4th CARDIS* (Bristol, UK, 2000), pp. 365–388.

[Gut00] GUTHERY, S. How to turn a GSM SIM into a web server. In *4th CARDIS* (Bristol, UK, 2000), pp. 209–224.

[GBPR00] GUTHERY, S., BAUDOIN, Y., POSSEGA, J., AND REES, J. Ip and arp over iso 7816-3. Internet Draft guthery-ip7816-00, 2000.

[HJF95] HARTEL, P. H., AND JONG FRZ, E. K. DE. Smart cards and card operating systems. Tech. rep., Dept. of EE and CS, University of Southampton, UK, 1995.

[ISO7816] INTERNATIONAL ORGANISATION FOR STANDARDISATION (ISO), JTC 1/SC 17. ISO/IEC 7816 Identification cards - Integrated circuit(s) cards with contacts.

[IFH00] ITOI, N., FUKUZAWA, T., AND HONEYMAN, P. Secure internet smartcards. In *1st JAVACARD* (Cannes, France, 2000), I. Attali and T. Jensen (Eds.), LNCS 2041, Springer-Verlag, pp. 73–89.

[KRV00] KEHR, R., ROHS, M., AND VOGT, H. Issues in smartcard middleware. In *1st JAVACARD* (Cannes, France, 2000), I. Attali and T. Jensen (Eds.), LNCS 2041, Springer-Verlag, pp. 90–97.

[RH00] REES, J., AND HONEYMAN, P. Webcard: A java card web server. In *4th CARDIS* (Bristol, UK, 2000), pp. 197-208.

[Sun99] SUN. Java remote method invocation specification. Tech. rep., Sun Microsystems, Inc., 1999. Revision 1.7.

Provably Secure Chipcard Personalization or How To Fool Malicious Insiders

Helena Handschuh[1], David Naccache[1], Pascal Paillier[1], Christophe Tymen[1,2]

[1] Gemplus Card International, Cryptography Group
34 rue Guynemer, 92447 Issy-les-Moulineaux, France
{helena.handschuh, david.naccache, pascal.paillier}@gemplus.com
[2] École Normale Supérieure, 45 rue d'Ulm, 75230 Paris, France
christophe.tymen@gemplus.com

http://www.gemplus.com/smart/

Abstract. We present 'malicious insider attacks' on chip-card personalization processes and suggest an improved way to securely generate secret-keys shared between an issuer and the user's smart card. Our procedure which results in a situation where even the card manufacturer producing the card cannot determine the value of the secret-keys that he personalizes into the card, uses public key techniques to provide integrity and privacy of the generated keys with respect to the complete initialisation chain. Our solution, which provides a non-interactive alternative to authenticated key agreement protocols, achieves provable security in the random oracle model under standard complexity assumptions. Our mechanism also features a certain genericity and, when coupled to a cryptosystem with fast encryption like RSA, allows low-cost intrusion-secure secret key generation.

1 Introduction

Tamper-resistant devices like smart-cards are used to store and process secret and personal data. Examples of applications making extensive use of smart cards include wireless communication systems such as the Global System for Mobile communications (GSM), or banking systems using the EMV (Europay, Mastercard and VISA) standard. These applications share the fact that they use secret key identification or authentication to achieve security and enable access to services. Thus some unique secret key K_I (we will adopt the notation K_I to denote a card's secret key by analogy with the widely known GSM terminology) needs to be shared between the issuer (the bank or the telecom operator) and the smart card. Usually this secret key material is downloaded into the card during the so-called chip personalization phase, i.e. the initialisation phase during which identical cards are configured in such a way that each and every of them corresponds to one specific user.

Usually, the card personalization center either writes secret keys into the cards according to a list provided by the issuer, or generates the keys itself and downloads them into the cards within it's own premises, and subsequently transmits a list of (encrypted) keys to the issuer. We refer to these scenarios as *typical* personalization protocols. In the sequel, we consider precisely the second scenario (key generation within the manufacturer's premises) and show that such a basic personalization procedure is vulnerable to *malicious insiders*.

We first discuss the potential security flaws in such a process, and then proceed to present a new personalization protocol in which the manufacturer is able to *provide evidence* to the issuer that no one except the issuer himself knows the secrets stored inside the cards.

Thus our new technique provides generally trusted keys for secret key applications.

The rest of the paper is organized as follows. In Section 2, we give an overview of a typical personalization protocol, and we point out its vulnerability to insider attacks when appropriate physical site protection measures are not enforced. Section 3 proposes our new personalization procedure. We provide a thorough security analysis in section 4 and conclude by a giving practical implementations of our technique in section 5.

2 Personalization Protocols

2.1 The current approach: typical protocols

Card personalization involves three parties: an issuer (telecommunications operator or bank), a card manufacturer (who actually personalizes smart cards for the issuer), and a smart card. Beyond graphical personalization – which may consist in printing the issuer's logo on the card for instance, the manufacturer has to electrically initiate the card and among such tasks, initialize the files meant to contain the card's secret key material K_I. In a typical scenario, each and every secret key K_I is generated uniformly at random by a personalization computer (PC) connected to the personalization system (such as a DataCard 9000 machine). Whenever a card enters the system, a fresh random key K_I is selected by the PC and downloaded into the card's non-volatile memory.

Simultaneously, the key gets encrypted on the PC, together with the card identifier Id (which might be some publicly available unique bitstring such as a serial number for instance) using the issuer's secret key K_s. Lists of encrypted (K_I, Id) pairs are then sent over an insecure channel to the issuer who decrypts the received files and recovers the pairs (K_I, Id).

Another way to proceed consists in encrypting the generated keys with the issuer's authenticated public key in an asymmetric key setting. This way, the issuer is the only entity able to decrypt the generated files, and the key K_s need not be known at the manufacturer's premises.

However, both solutions are vulnerable to insider attacks where a malicious entity having access to the manufacturer's premises would get hold of the key. We may think of a malicious insider as some malevolent employee willing to clone SIM cards or as a hacker that discretely eavesdrops the computer network from outside the personalization center. This strongly motivates the search for protocols featuring a guaranteed level of security against this kind of threat.

2.2 Security Notions for Card Personalization

Let us examine the setting and determine which security goals are desirable to reach from the issuer's standpoint. When the personalization protocol takes place, parties are

- a tamper-resistant secret-less chip-card to be personalized with identifier Id,
- an issuer (supposedly remote),
- a personalization system (PC + DC9000) in which the issuer has no reason to put trust.

Ultimately, the goals the personalization protocol is meant to achieve are the following. At the end of the process,

1. the card must contain some secret key K_I belonging to some fixed key space (we call this property *correctness*),
2. the issuer must know the correct pair (Id, K_I) (we refer to this property as *key integrity*),
3. the issuer should be confident that he is the only entity who shares the knowledge of the (Id, K_I) pair with the card (this is defined as *key privacy*).

Correctness is easily achieved. The question is whether requirements 2 and 3 are actually achieved by current typical personalization protocols, and the answer is obviously no. The above protocols do not meet key integrity nor even key privacy. Indeed, the computer, if handled by a malicious person, may very well generate a given K_I and transmit a different one to the issuer. This can be considered a denial of service attack, as the end-user would get a non-functional card. Alternatively, the computer might respect the integrity property by providing the right pair (Id, K_I) to the issuer, but reveal this pair to an intruder getting hold of the PC. In this case, card cloning becomes possible. We call such attacks 'malicious insider attacks'.

2.3 The Interlock protocol

One obvious attempt to address this problem consists in executing a key agreement protocol such as *Interlock* [4] between the card and the issuer.

Interlock is described as follows. Assuming that two entities A and B, with public-keys pk_A and pk_B, want to exchange a secret through an insecure channel, A and B proceed as follows. First, A and B exchange their public keys through the channel. Then, A (resp. B) chooses a random r_A (resp. r_B), and encrypts it with pk_B (resp. pk_A) to obtain a ciphertext c_A (resp. c_B). c_A (resp. c_B) is a bitstring which can be cut into two equal parts (c_A^1, c_A^2) (resp. (c_B^1, c_B^2)). Thus, A sends c_A^1 to B, and sends the remaining part c_A^2 only after having received c_B^1. Finally, B sends c_B^2. At the end of the sequence, A and B share the pair (r_A, r_B).

Clearly, this protocol thwarts passive man-in-the-middle attacks. However, it is interactive, which represents an unacceptable hurdle in the context of a personalization process. The only way to achieve an equivalent non-interactive protocol would be to use public-key certificates and signature verification which calls for far too complex (and heavy) public-key infrastructures.

Besides, security requirements explicitly demand resistance against active attacks, where the attacker may not only eavesdrop exchanged pieces of information but also modify them in some way, and may impersonate parties as well. Because it does not provide authentication, Interlock does not resist active attacks.

The contribution of this paper consists in providing a non-interactive alternative to the Interlock protocol which, in our context, resists active attacks and needs no certificates or signatures whatsoever.

3 A Provably Secure Card Personalization Protocol

Let us go back to the typical scenario. Obviously, the security breach resides in the possibility to attack the PC. Thus each and every secret should be generated *inside the card itself*, which, by assumption, provides the advantage of being tamper-resistant over an open PC.

3.1 A First approach

Thus a first idea is to generate the secret key K_I inside the card, download the issuer's public key into the card, encrypt the generated secret under the public key and output the result. Next, the encrypted secrets are collected along with the Id's in a file and sent to the issuer who decrypts the list with his private key SK and recovers the associated pairs in clear (alternatively the Id's could also be encrypted together with the secret K_I inside the card). This protocol is shown in figure 1. The public key of the issuer is noted PK; typically, one could use stand-alone RSA public key encryption [5] for instance. We suppose the key pair (PK, SK) is generated once and for all by the issuer himself, and then transmitted to the personalization center which uses it for a certain period of time.

Unfortunately, this solution is vulnerable to the well-known man-in-the-middle attack. Suppose the attacker controls the PC again. She is then able to generate her own public

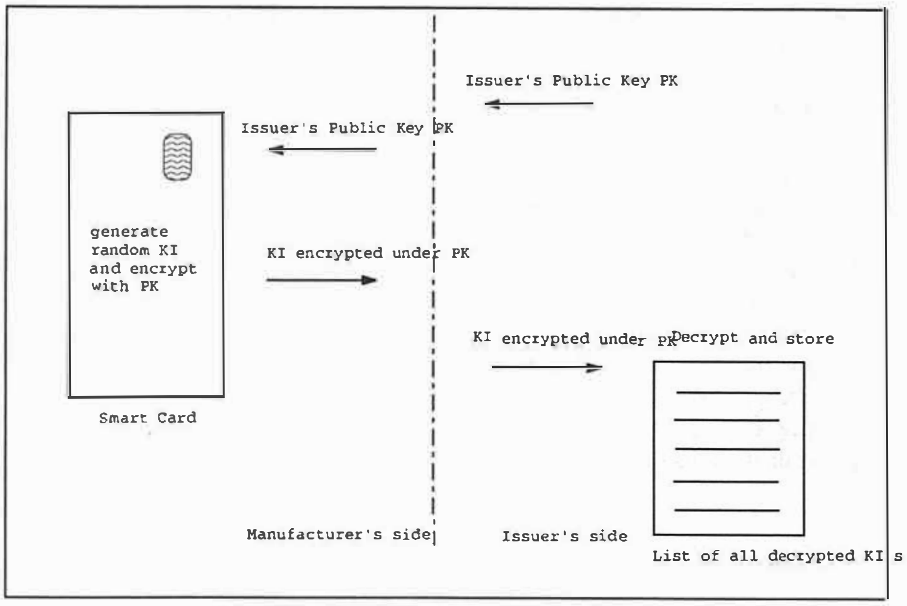


Fig. 1. Secure personalization protocol : first approach

and private RSA key pair and to fool the card by sending to it her own public key. She recovers the encrypted K_I values, decrypts them, and re-encrypts them with the issuer's public key. Thus key integrity is preserved, but key privacy is violated. The attack is shown in figure 2 where the attacker's public key is noted PK'.

3.2 Proposed Protocol

Let us now proceed to describe our protocol. The security analysis will be discussed in the next section. Basically, the personalization process now includes the following steps:

1. the PC transmits the issuer's public key PK to the card,
2. the card generates a random r, computes $K_I = H(r, \mathsf{PK})$ where H is a hash function such as SHA-1 [7], and memorizes K_I in non volatile memory,
3. the card encrypts r as $c = \mathcal{E}_{\mathsf{PK}}(r)$ where $\mathcal{E}_{\mathsf{PK}}$ denotes public key encryption under PK, and outputs c,
4. the PC collects the pair (Id, c) and sends it to the issuer who later decrypts c using SK, recovers $r = \mathcal{D}_{\mathsf{SK}}(c)$ and computes $K_I = H(r, \mathsf{PK})$.

This protocol is shown in figure 3.

4 Security Analysis

4.1 Main Results

Although looking simple, our protocol achieves a very satisfactory security property, namely that

- both key integrity and key privacy are preserved under a passive attack,
- if key privacy is not preserved under an active attack then key integrity cannot be preserved either.

The proof of that fact is given below. From a practical viewpoint, this means that if an intruder simply eavesdrops what is transmitted through the PC, our protocol fully reaches the security goals of section 2.2, namely key integrity and privacy. Additionally, if the intruder actively operates changes over transmitted data, she is given no other choice than

- either knowing the key K_I generated by the card; but then the issuer recovers

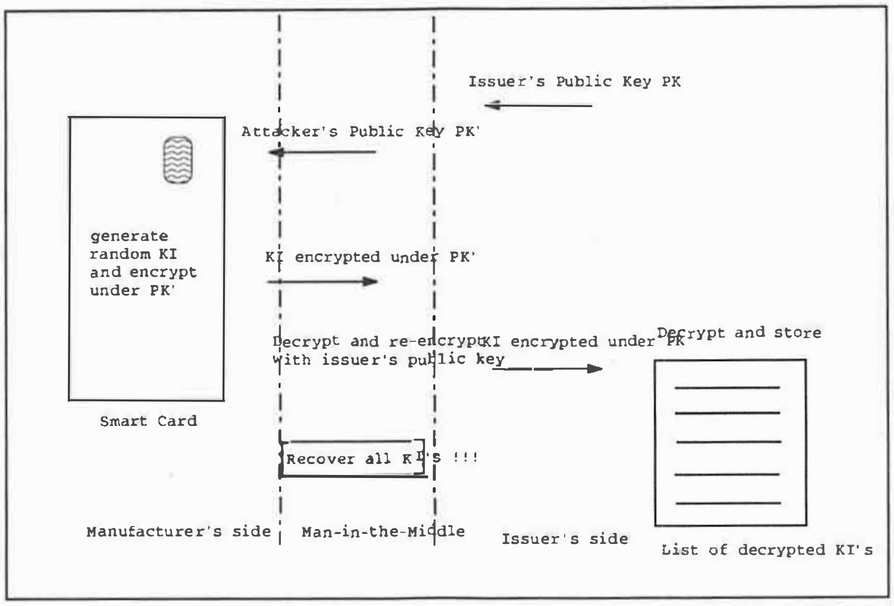


Fig. 2. Man-in-the-middle attack on key generation process

nothing else than a faulty key $K'_I \neq K_I$. Subsequently, the card just cannot work properly because user authentication will be unsuccessful each time the end user attempts to access the issuer's service. The issuer may then recognize the card as a fake or abnormal one and blacklist it.

- or letting the card generate K_I properly and later have normal access to the issuer's service; but then, no information whatsoever can be obtained on K_I.

In other words, our protocol prevents insiders from cloning normal cards since only useless cards are exposed to key divulgation. Trying to gain information on the card's key simply forbids its future use in normal conditions. We guarantee this under any type of attacks, be they very sophisticated. The insider is left only with malevolence i.e. the ability to force the personalization of useless cards. We argue that this scenario is not of interest to an active adversary. We assess these results without considering collusions in the first place, and address these further in section 4.5.

4.2 Security Proof Against Passive Insiders

We state, in a somewhat more formal way:

Theorem 1 (Passive Attacks). *Assume that the encryption scheme $\mathcal{E}_{\mathsf{PK}}$ is deterministic and one-way under chosen plaintext attacks (OW-CPA). Then no polynomial time attacker given* PK *and $c = \mathcal{E}_{\mathsf{PK}}(r)$ can recover $K_I = H(r, \mathsf{PK})$ with non-negligible probability in the random oracle model.*

Proof. We assume the existence of an attacker $\mathcal{A}$ with success probability ϵ and show how to invert $\mathcal{E}_{\mathsf{PK}}$ with probability ϵ'. We build a reduction algorithm $\mathcal{B}$ as follows. $\mathcal{B}$ is given an instance $\tilde{c} = \mathcal{E}_{\mathsf{PK}}(\tilde{r})$ and must return $\tilde{r}$ with non-negligible probability. $\mathcal{B}$ randomly selects $\widetilde{K_I}$ and runs $\mathcal{A}(\mathsf{PK}, \tilde{c})$.

Now, each time $\mathcal{A}$ queries the random oracle H for an input (r,pk), $\mathcal{B}$ checks in the history of queries if (r,pk) was queried by $\mathcal{A}$ in the past, in which case the same answer is returned to $\mathcal{A}$. Otherwise, if $pk = \mathsf{PK}$ and $\mathcal{E}_{\mathsf{PK}}(r) = \tilde{c}$, then $\mathcal{B}$ sets $\tilde{r} = r$ and returns $\widetilde{K_I}$. If none of these cases occur, $\mathcal{B}$ selects h uniformly at random, returns h and updates the

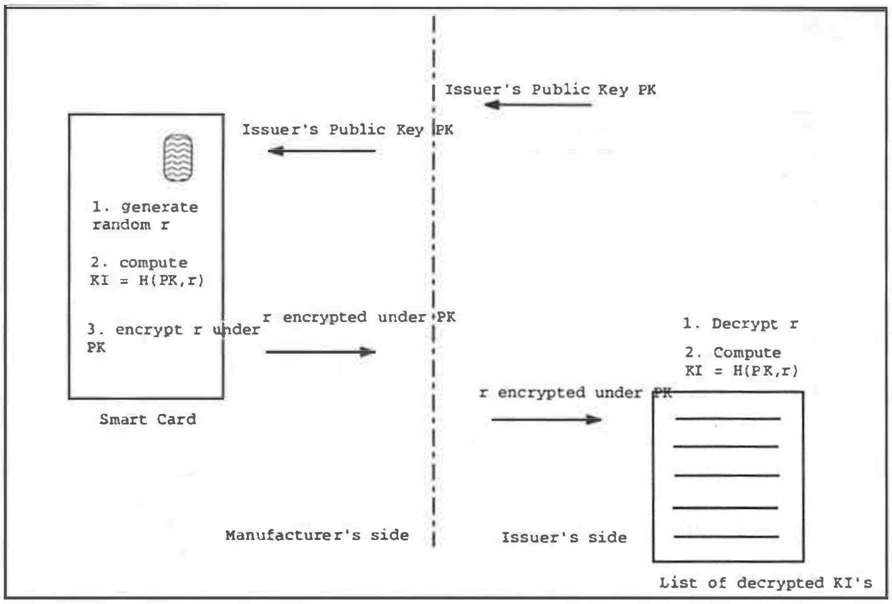


Fig. 3. Provably Secure Card Personalization Protocol

history of queries. Now when $\mathcal{A}$ has finished, $\mathcal{B}$ checks whether $\widetilde{r}$ was initialized during the game, simply returns $\widetilde{r}$ if so or fails otherwise. This completes the description of the reduction algorithm $\mathcal{B}$.

Since the simulation of H is perfect, it is clear that B is sound. We denote by Ask the event that $\mathcal{A}$ submits $\widetilde{r}$ to the simulation of H. Now if Ask never happens, $\widetilde{K_I}$ is a uniformly distributed random value unknown to $\mathcal{A}$, so

$$\Pr\left[\mathcal{A} = \widetilde{K_I} \mid \neg\mathsf{Ask}\right] \leq \frac{1}{\sharp H},$$

where $\sharp H$ denotes the number of elements in the output space of H. By assumption,

$$\begin{aligned}\epsilon &\leq \Pr\left[\mathcal{A} = \widetilde{K_I}\right] \\ &\leq \Pr\left[\mathcal{A} = \widetilde{K_I} \mid \neg\mathsf{Ask}\right] + \Pr[\mathsf{Ask}] \\ &\leq \frac{1}{\sharp H} + \Pr[\mathsf{Ask}]\end{aligned}$$

which yields

$$\begin{aligned}\epsilon' &= \Pr[\mathcal{B} = \widetilde{r}] \\ &= \Pr[\mathsf{Ask}] \\ &\geq \epsilon - 1/\sharp H\end{aligned}$$

Therefore, if ϵ is non negligible, ϵ' is non negligible either. □

Interestingly, we also get a slightly different result for *non deterministic* encryption schemes, i.e. when the protocol relies on a probabilistic encryption function $r \mapsto \mathcal{E}_{\mathsf{PK}}(r; u)$. We include this result here for the sake of completeness. We state:

Theorem 2 (Passive Attacks). *Assume that the probabilistic encryption scheme $\mathcal{E}_{\mathsf{PK}}$ is semantically secure under chosen plaintext attacks (IND-CPA). Then no polynomial time attacker given* PK *and $c = \mathcal{E}_{\mathsf{PK}}(r)$ can recover $K_I = H(r, \mathsf{PK})$ with non-negligible probability in the random oracle model.*

Proof. Here again, we assume the existence of the same attacker $\mathcal{A}$ with non negligible success probability ϵ and show how to distinguish encryptions $\mathcal{E}_{\mathsf{PK}}$ with non negligible advantage ϵ'. The reduction algorithm $\mathcal{B} = (\mathcal{B}_1, \mathcal{B}_2)$ is as follows. $\mathcal{B}_1$ (the find stage) chooses two distinct messages (r_0, r_1) uniformly at random and outputs them. Then $\mathcal{B}_2$ inputs $c_b = \mathcal{E}_{\mathsf{PK}}(r_b; u)$ for a certain bit b and random tape u. $\mathcal{B}$ must guess b with non negligible advantage.

To do this, $\mathcal{B}_2$ is designed as follows. $\mathcal{B}_2$ randomly selects $\widetilde{K_I}$ and runs $\mathcal{A}(\mathsf{PK}, c_b)$. Each time $\mathcal{A}$ queries the random oracle H for an input (r,pk), $\mathcal{B}_2$ checks in the history of queries if (r,pk) was queried by $\mathcal{A}$ in the past, in which case the same answer is returned to $\mathcal{A}$. Otherwise, if $pk = \mathsf{PK}$ and $r = r_b$ for $b \in \{0,1\}$, then $\mathcal{B}_2$ stops and output b. If none of these cases occur, $\mathcal{B}$ selects h uniformly at random, returns h and updates the history of queries. If $\mathcal{A}$ finishes, $\mathcal{B}$ stops, chooses $\beta \in \{0,1\}$ at random and returns β. This completes the description of the reduction algorithm $\mathcal{B}$.

The simulation of H is almost perfect. We denote by Good the event that $\mathcal{A}$ submits r_b to the simulation of H and by Bad the event that $\mathcal{A}$ submits $r_{\bar{b}}$ to the simulation of H. Now if neither Good nor Bad ever happens, $\widetilde{K_I}$ is a uniformly distributed random value unknown to $\mathcal{A}$, so

$$\Pr\left[\mathcal{A} = \widetilde{K_I} \mid \neg(\mathsf{Good} \vee \mathsf{Bad})\right] \leq \frac{1}{\sharp H} .$$

By assumption,

$$\begin{aligned}\epsilon &\leq \Pr\left[\mathcal{A} = \widetilde{K_I}\right] \\ &\leq \Pr\left[\mathcal{A} = \widetilde{K_I} \mid \neg(\mathsf{Good} \vee \mathsf{Bad})\right] \\ &+ \Pr\left[\mathsf{Good} \vee \mathsf{Bad}\right] \\ &\leq \frac{1}{\sharp H} + \Pr\left[\mathsf{Good} \vee \mathsf{Bad}\right] .\end{aligned}$$

Since the choice of (r_0, r_1) is independent from $\mathcal{A}$'s view, the probability that $r_{\bar{b}}$ is submitted by $\mathcal{A}$ to the random oracle H is upper bounded by $1/\sharp r$. Given that Good and Bad exclude each other, we get

$$\begin{aligned}\Pr\left[\mathsf{Good}\right] &= \Pr\left[\mathsf{Good} \vee \mathsf{Bad}\right] - \Pr\left[\mathsf{Bad}\right] \\ &\geq \Pr\left[\mathsf{Good} \vee \mathsf{Bad}\right] - \frac{1}{\sharp r} .\end{aligned}$$

Therefore

$$\begin{aligned}\frac{1+\epsilon'}{2} &= \Pr\left[\mathcal{B} = b\right] \\ &= \Pr\left[\mathsf{Good}\right] \\ &+ \Pr\left[\neg(\mathsf{Good} \vee \mathsf{Bad}) \wedge \beta = b\right] \\ &= \Pr\left[\mathsf{Good}\right] + \frac{1}{2}\Pr\left[\neg(\mathsf{Good} \vee \mathsf{Bad})\right] \\ &= \frac{1}{2} + \Pr\left[\mathsf{Good}\right] - \frac{1}{2}\Pr\left[\mathsf{Good} \vee \mathsf{Bad}\right] \\ &\geq \frac{1}{2} + \frac{1}{2}\Pr\left[\mathsf{Good} \vee \mathsf{Bad}\right] - \frac{1}{\sharp r} \\ &\geq \frac{1}{2} + \frac{1}{2}(\epsilon - \frac{1}{\sharp H}) - \frac{1}{\sharp r} ,\end{aligned}$$

and finally $\epsilon' \geq \epsilon - 1/\sharp H - 2/\sharp r$ as wanted.

□

4.3 Security Proof Against Active Insiders

We now focus on security against active insiders. We have:

Theorem 3 (Active Attacks). *Assume the encryption scheme $\mathcal{E}_{\mathsf{PK}}$ is deterministic and one-way or probabilistic and semantically secure (under chosen ciphertext attacks). Then obtaining information about K_I requires the attacker to corrupt the value of* PK. *Then $K_I \neq H(r, \mathsf{PK})$ with overwhelming probability.*

Proof. Essentially, we follow the initial work of [6]. Suppose indeed, that the attacker does not alter the value of PK which is transmitted to the card. Two situations may occur:

1. either the insider corrupts the value of $c = \mathcal{E}_{\mathsf{PK}}(r)$ by changing it into c', but this of of no use whatsoever to her,
2. or she does not corrupt c; in this case, the insider is passive and theorem 1 or 2 applies, depending on $\mathcal{E}_{\mathsf{PK}}$. This means that no information about K_I can be obtained.

On the other hand, if the insider controls the PC and cheats on PK, she may recover

K_I by submitting another public key PK' but the issuer then gets a different value $H(r, \mathsf{PK}) \neq H(r, \mathsf{PK}')$ with overwhelming probability. Thus the card will not be functional and no damage (other than denial of service) will incur to the issuer. This provides evidence that either the protocol is correct, or the card will not function at all. □

4.4 Can Denial of Service Be Avoided?

What is desirable is that the protocol would preserve both key integrity and privacy under any attack circumstances, as this would thwart denial of service attacks discussed above. For theoretical reasons, however, no protocol can achieve such a better security level without an authenticated communication channel between the card and the issuer. The only cheap way to achieve authentication would consist in masking the issuer's public key PK into the read only memory (ROM) of the card. We would then reach both key integrity and privacy in any case. But we recall that denial of service does not serve the attacker's interests anyway because it precisely testifies the presence of an active attack during the personalization process.

4.5 Collusion attacks

An intuitive way to break the system would be to envision the collusion between a malicious insider and a malicious issuer. For example, the insider might substitute the genuine issuer's public key with the malicious issuer's public key. In this case, under the unusual assumption that both issuers use the same operating system on the card, the personalized cards would work on the malicious issuer's network whereas they would *not* work on the genuine network. Although this scenario theoretically exists, one cannot help wondering what benefit the malicious issuer could possibly get out of this setting. First, the cards are shipped to the initially intended recipients or more generally speaking directly to the user. Thus the malicious issuer will never get hold of the cards. Second, this issuer would then have cards in the field that can and will be used on his own network, but he could not plausibly recover any fees associated to this usage. So the users would simply (say) use wireless communication networks without paying a dime to the malicious operator.

Interestingly, we could also envision attacks combining an active intrusion with a partial or total access to the issuer's decryption server. This would allow the attacker to query the server for r-values of her choice given c, possibly excepting the ones that correspond to already listed K_I's (as this could cause some kind of collision detection by the server). This is exactly a chosen-ciphertext attack scenario and in this case, again, our protocol remains fully secure in the same sense, provided that the underlying encryption scheme $\mathcal{E}_{\mathsf{PK}}$ be OW-CCA or INC-CCA (instead of OW-CPA or IND-CPA). This is easily obtained as a natural extension of theorems 1 and 2. Then chosen-ciphertext secure encryption schemes like RSA-OAEP [1] or Cramer-Shoup [2] must be employed.

A denial of service attacker can always interact with the chip-card in such a way that in the end the card is invalid. But, as stressed before, we assume that this scenario is not of interest to an active adversary. We also stress the fact that more elaborate attacks where the complete set of employees of the manufacturer collude against the issuer are not considered in this paper. As an illustration, these include situations where the card's operating system itself is flawed or corrupted and does not fully respect the protocol.

In light of the above discussion, we believe that no other protocol can further enhance the one we propose in this setting, except if additional key authentication is implemented in some way or an other.

5 Practical Examples

5.1 An Example Using Low-Exponent RSA

We recall the protocol steps in this context, taking SHA-1 as an embodiment of H. First, the issuer generates an RSA key pair $(\mathsf{PK}, \mathsf{SK})$ where $\mathsf{PK} = (n, e)$ and $\mathsf{SK} = (n, d)$ with $n = pq$ for two large primes p and q, $e = 3$ for instance and $d = e^{-1} \mod (p-1)(q-1)$ (RSA key generation imposes that $\gcd((p-1)(q-1), e) = 1$). The manufacturer is given n and for each card to be personalized, engages the PC in the following protocol:

1. the PC transmits n to the card with identifier Id,
2. the card selects r uniformly at random and computes $K_I = \text{SHA-1}(r, n)$,
3. the card computes $c = r^3 \mod n$ and outputs c,
4. the PC collects the pair (Id, c) and sends it later to the issuer,
5. the issuer recovers $r = c^d \mod n$, computes $K_I = \text{SHA-1}(r, n)$ and stores the pair (Id, K_I).

Note that this is extremely efficient, as the card only performs a couple of modular multiplications and a single call to SHA-1. Moreover, we have the following security statement.

Corollary 1 (of theorems 1 and 3). *Assuming the random oracle model, under the RSA assumption, malicious insiders cannot retrieve the secret key K_I of a functional card.*

5.2 An Example Based on the Diffie-Hellman Problem

It is possible to adapt the above protocol in order to use the Decision Diffie-Hellman (DDH) as the underlying intractability assumption. This is done by choosing El-Gamal encryption [3] to instantiate $\mathcal{E}_{\mathsf{PK}}$ instead of RSA, as follows.

The issuer chooses an abelian group G, denoted multiplicatively, of large order q, in which the discrete logarithm is intractable. An elliptic curve defined over a finite field, or the group of integers modulo a large prime p are examples of such a group. The issuer then chooses a base $g \in G$, a random integer $1 < x < q$, stores $\mathsf{SK} = x$ and transmits $\mathsf{PK} = (g, g^x) := (g, h)$. The personalization process now works as follows:

1. the PC transmits the issuer's public-key (g, h) to the card with identifier Id,
2. the card selects r uniformly at random and computes the pair (g^r, h^r),
3. the card computes $K_I = \text{SHA-1}(h^r, g, h)$, memorizes K_I in non-volatile memory and outputs g^r,
4. the PC sends the pair (Id, g^r) to the issuer, who later recovers K_I by computing $K_I = \text{SHA-1}((g^r)^x, g, h)$.

In this case, we get the following security result.

Corollary 2 (of theorems 2 and 3). *Assuming the random oracle model, under the DDH assumption, malicious insiders cannot retrieve the secret key K_I of a functional card.*

6 Conclusion

We have presented a simple provably secure protocol which enables a smart-card manufacturer to act as a trusted personalization center without knowing any secret data belonging to the issuer. The proposed solution does not require a public-key infrastructure, and avoids all the secret-key management procedures usually required to guarantee the security of the personalization process.

Acknowledgments

We thank Jacques Stern for his help on the initial version of the security proofs and anonymous reviewers for their comments and suggested improvements of the paper.

References

1. M. Bellare and P. Rogaway. Optimal asymmetric encryption. In A. De Santis, editor, *Advances in Cryptology - EUROCRYPT '94*, volume 950 of *Lecture Notes in Computer Science*, pp. 92–111, Springer-Verlag, 1995.
2. R. Cramer and V. Shoup. A Practical Public-Key Cryptosystem Provably Secure Against Adaptive Chosen-Ciphertext Attacks. In *Advances in Cryptology - CRYPTO '98*, volume 1462 of *Lecture Notes in Computer Science*, pp. 13–25, Springer-Verlag, 1998.
3. T. El Gamal. A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms In IEEE Trans. Inform. Theory, vol. 31, pp. 469–472, 1985.
4. R. Rivest and A.Shamir, How to expose an Eavesdropper, Communication of the ACM, v.27, n.4, Apr. 1984, pp. 393–395
5. R. Rivest, A. Shamir and L. Adleman, A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. In Communications of the ACM, vol. 21, n 2, p.120–126, February 1978.
6. J. Stern, Analysis of a Secure Chipcard Personalization Protocol. Unpublished Manuscript, January 2001.
7. US Department of Commerce, N.I.S.T. Secure Hash Algorithm. In FIPS 180-1, 1995.

Automatic Code Recognition for smart cards using a Kohonen neural network

Jean-Jacques Quisquater
David Samyde
Université catholique de Louvain, UCL Crypto Group
Place du Levant, 3, B-1348 Louvain-la-Neuve, Belgium
{jjq,samyde}@dice.ucl.ac.be, http://www.dice.ucl.ac.be/crypto

Abstract

A processor can leak information by different ways. Although, the possibility of attacking smart cards by analyzing their power consumption [Kocher] or their electromagnetic radiations is now commonly accepted [Gandolfi]. A lot of publications recognize the possibility to recover the signature of an instruction in a side channel trace. It seems that no article demonstrate how to automate reverse engineering of software code, using this assumption. Our work describes a method to recognize the instructions carried out by the processor. In a general way, a classifier permits to identify the right or wrong value during the comparison of a pin code or large parts of a software code. On a few micro-controllers, using a classical correlation between the power trace and a dictionary, we show how to identify the CPU's actions. Sometimes, silicon manufacturers hide specific opcodes deliberately. The EM investigation and the template attack demonstrated by IBM, at Cryptographic Hardware and Embedded Systems 2002, rely on multivariate signal processing for electromagnetic and power traces. The method presented in this article is based on a self organizing map. On a CISC processor, it is then obvious to find a hidden instruction looking for a hole or a bad construction of the map. The case of pipelined processors is a little bit different: as they decode, execute, fetch, several parts of different opcodes at the same time, it is more difficult to recognize a specific signature.

1 Introduction

Processor's power consumption has been known for a long time by a restricted group of people in the smart card community. For security purposes, the knowledge of this side channel has been kept secret. In 1998, Paul Kocher introduced the concept of Differential Power Analysis. It was the first introduction of real signal processing for smart card attacks... Differential Power Analysis (DPA) can be explained as the correlation of two random variables. Today, people are confident with power analysis, but they also know that current measurement is not the only source of information leakage. The electromagnetic radiations can give the same result. Last year, the security group of Gemplus demonstrated that electromagnetic side channel must be taken into account seriously [Gandolfi]. The signal noise ratio of electromagnetic analysis (EMA) is much better than the signal noise ratio of power analysis. EMA measurements are very noisy. Power analysis measurements contain lower frequencies than EMA. For the same processor, the number of traces necessary to recover the secret key is reduced for EMA. The practical implementation of power analysis is very simple to realize unlike EMA. One of the big advantages of EMA is the locality principle. Using a very little sensor, it is obvious to localize exactly a specific leakage source on a processor [Quisquater]. Actual improvements of classical non intrusive analysis are linked to sophisticated signal processing [Boneh]. For differential analysis, Bayesian methods allow to recover the secret key with much less than forty traces.

On of the most important parts in differential side channel analysis, is the decision criteria. A classical countermeasure against power or electromagnetic analysis is to defeat the selection function of the attacker. In this case wrong guesses appear. An engineer very familiar with power trace of a specific processor can easily recognize each instruction carried out by the device. Thus it is possible to build a tool dedicated to parse the trace and to gather opcodes during a simple acquisition. A neural network can improve the decision criteria. With such a tool, once the learning phase done, it must be easy to recover instructions.

Two instructions executed on a Complex Instruction Set Computer processor can necessitate a different number of clock cycles. Generally a Reduced Instruction Set Computer processor execute one instruction per clock cycle. The number of clock cycles per instruction is a source of information for an attacker. Asynchronous processors do not have clock, so their actions are not easily identifiable. Anyway, it is possible to recover very specific patterns such as some memory access (charge pump) parsing power or EM traces. For a classical processor, the role of the clock is dominating, and the principal component of consumption remains is linked to the clock signal.

2 Chip depackaging

For a long time, smart cards have been famous for their tamper resistance [Anderson]. They are protected against invasive attacks. Security sensors are various, so a lot of attacks can be stopped very early. In order to discourage attackers, it is quite difficult to open the package and directly access the microprocessor without damaging it. Many traps have been introduced by designers. A lot of processors are surrounded by grids, physical parameters are checked automatically. The continuity or the resistance of these sensors indicates a correct operation mode to the processor. But the engineers do not only use passive sensors, they also introduce false radiation sources, clock jitter, low temperature and light detectors...

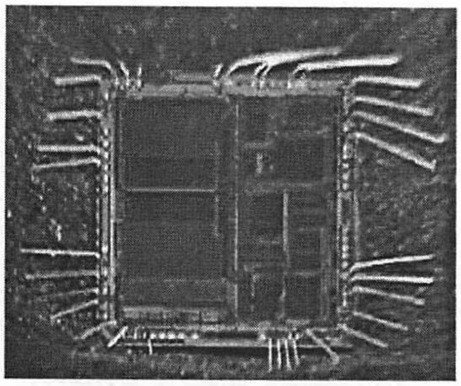


Figure 1: A depackaged processor

Recent attacks permit to cancel some sensors. With very concentrated nitric acid and organic solvent it seems to be possible to depackage a smart card or a classical processor without damaging it. When the chip is depackaged it is very easy to see regular structures or external sensors, sometimes without a specific microscope. Inside a structure, the logical gates are mixed in order to complicate the spot of a person wishing to do reverse engineering.

On figure 1, a classical processor has been opened. It is based on an eight bit architecture including a pipeline. All the instructions are done into four or eight clock cycles. On the left side, two identical banks are visible, and on the other side non regular structures are present. This pipelined processor is well known from the pay-TV pirates. It is currently extremely used with an I^2C memory for false smart cards. In general, test circuits of classical smart card are specific, it is not the case here.

3 Power measurements vs electromagnetic measurements

The electromagnetic radiations have recently been investigated as one of the new sources of side channel for the smart cards. The magnetic field is much more investigated than the electric one. The sensor is very often a coil, placed in the close field of the processor.

If we separate the spectrum into two principal components, each one dominating largely over a frequency band, the electric field from DC to 10 MHz carries information different from the magnetic field. In fact the propagated wave is not the same at all. A small capacitor or a wire simulating an antenna is enough to investigate seriously effects of the electric field. It allows to locate more precisely some parts of the chip (phase locked loop, charge pumps,). Inspecting this band also permits to recover the presence of the clock signals, often lower than 10 MHz. Some smart card integrate their own internal clocks or frequency multipliers.

The traditional power consumption analysis [Messerges] recovers the actions of the processor but does not permit to map a chip. The global consumption measurement is the sum of local consumption of each local substructures of the smart card. It may be possible with some signal processing to isolate the consumption from each component. We wish to retrieve the code executed by the processor. In order to be able to isolate the circuitry concerning instructions, we use electromagnetic radiation but particularly the electric field.

4 The pipe-line and the influence of each instruction

The processor we want to analyze contains a four-stage pipeline. Four clock cycles are necessary to the processor to carry out an instruction. But at each clock cycle it is in fact a fetch, a decode, an execution, a storage. The influence of the pipeline is then present on the side channel trace. And in an extended way, the preceding instruction modifies the consumption of the instruction in course of execution.

In order to highlight such an assertion, we have an instruction (A) executed by the processor followed by several identical instructions (Bi). This reveals that the first instruction (B1) after the instruction (A) has its trace modified by the instruction (A). We can realize here that the first quarter of the trace is stronger if the instruction before was a Bit Clear in File register (BCF) with a pad set to one.

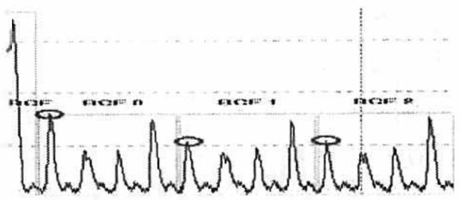


Figure 2: The interaction between two instructions

Instructions can have interaction with themselves. In order to validate our working hypothesis, we execute several identical instructions (Bi). Normally, all the instructions should have an identical trace. In fact it is not the case and the "action" of the first instruction modifies the trace of the second instruction. It is exactly what the figure 2 demonstrates.

The signature of each instruction contains four peaks. On figure 2, the first peak of the BCF is directly linked to the instruction just before, but not only. When BCF follows another BCF having called upon the establishment of an external pad, to one using a particular register, the first peak is different. Between the BCF 0 and the BCF 1, the difference is visible, and is only related to the external action of the preceding instruction.

5 An instruction signature

Each instruction gives a different trace by power analysis [Fahn]. But the signature of an instruction is an expression of its own address in memory, the data handled and sometimes the address where the data will be stored. The Hamming weight of each data/adress is clearly visible by electromagnetic analysis, but it is still impossible to detect 55 to a *AA* as their Hamming weight is the same. Using the electric field, it is possible to recover more information. Our antenna is based on a bonding wire, so if the

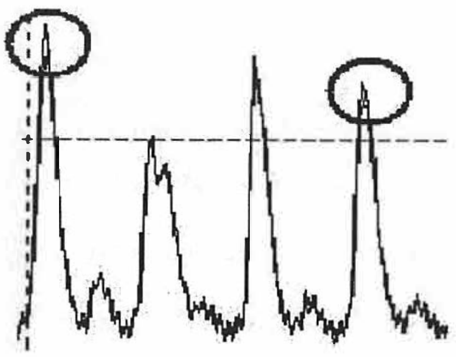


Figure 3: Two signatures with two addresses and two Hamming weight

architecture of the bus does not equilibrate the electric field radiated, it is possible being closer to a group of wire to identify their activity. A possible countermeasure against such an analysis is to use two wires for coding 0 and 1. It is called dual logic, two other states are free to specify alarms. (01 = 0, 10 = 1, 00 = 11 = alarm). Ross Anderson, Markus Kuhn and Sergei Skorobogatov, suggest to use this architecture to design new processors. If a pair of wire is set to 00 or 11 it products a detectable fault.

The influence of an instruction's address on its signature can be easily showed. The same couple of instructions executed several times at different addresses does not permit to obtain a constant trace. The Hamming weight involved is very different. The figure 3 shows the influence of the address on the first and last cycles of the clock. This figure represents two instructions CLear Work Register (CLRW) at two addresses with an extremely different Hamming weight. The first modification corresponds with the address execution of the current instruction, whereas the last peak is the expression of the address of the next instruction to be executed by the pipeline.

Moreover, consumption traces clearly indicate that the consecutive addresses sometimes contain an identical Hamming weight. Figure 4 details the trace of consumption of the instruction SUBstract Literal from Work register (SUBLW) for nine continuous addresses. The first clock cycle is influenced by the address of the instruction executed. Two consecutive address with the same Hamming weight generate two identical power peaks.

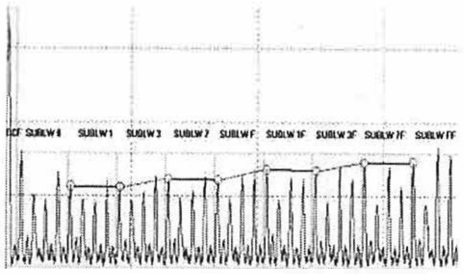


Figure 4: The influence of the address on the signature

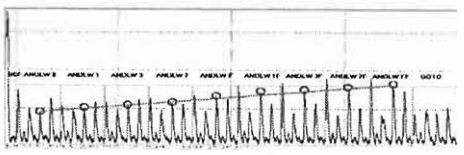


Figure 5: The influence of data's Hamming weight

The Hamming weight of the manipulated data is important too [Kommerling]. On our processor, the third peak expresses this data. Figure 5 proves that quite a linear equation between the Hamming weight and the consumption exists. Of course the processor under analysis does not contain any power or electromagnetic analysis countermeasures. Advanced smart card processors are generally protected against non intrusive analysis [Coron]. In some processors the data bus is encrypted, (sometimes the address bus too). If an attacker is able to recognize patterns using the Hamming weight, he can be able to extract cryptographic keys [Kuhn]. Then to avoid such an analysis, the implementation cryptographic algorithm are well secured against power or electromagnetic analysis.

Transfering address modifies the shape and the very last peak of an instruction's power trace. The linear properties are maintained compare to the address properties. The main difference between the influence of the Hamming weight on address and data is the localization in the signature.

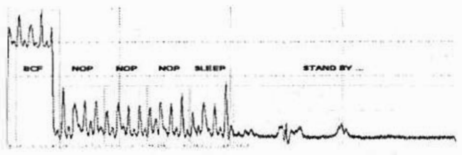


Figure 6: The sleep instruction

6 A correlation attack

The processor we decided to investigate can be described with only one template for all its instructions. But some processors do not have instructions with quite the same signature. In a few cases, the power traces [Biham] are very different for each instruction compared to the others, and their automatic recognition is common place.

In our case, for the processor studied, each instruction has a signature on a four clock cycles. On the other hand, the values of the various peaks are directly related to the opcode. It is the case of figure 6. The sleep instruction gives a characteristic signature just before the deactivation of the processor.

It is then possible to carry out a recognition of the instructions using a simple correlation with a dictionary . Building a dictionary where each instruction is represented by t_n points of measurement is not so difficult. Each power or EM trace is saved in the dictionary as S_n points. By multiplying each t_i point with the S_i corresponding value, and by normalizing the result, we obtain a specific opcode detector. Standardization is not necessary because if measurements are carried out in the same environment as during the creation of the dictionary, the voltages and the measured currents are the same too. The only requirement is to have a comparator triggered on a high threshold, in order to secure the false alarms.

The calculation of correlation avoids the permanent resynchronization. To isolate the known patterns from a consumption trace, it is necessary to be synchronous with the trace [Kelsey]. Our experiments for the moment are not executed in real time, and our traces are stored over for a long time by using a memory board of 1 Gigabytes. Then the values go trough a correlator which provides after a comparison/threshold the name of the instruction identified with the Hamming weight concerned. This method works with a high level recognition (better than 87%) better for CISC processor than for RISC ones. Anyway we tested it on a Z80 processor and we managed to recover more than 95% of a software. Parsing the power and EM trace, we were able to extract each pattern and to identify it, using the dictionary built just before.

Ideally we wish to be able to find the codes carried out on RISC processors, but as many instructions have very close signatures, the error rate becomes important.

7 Self organizing maps

It is then necessary to change the structure we used. We have decided to use an automatic classifier. This work can be done by a neural network.

The Kohonen's self organizing maps are based on a network of K neurons with N input. The network has K outputs. The inputs are vectors with N components, all connected completely to K neurons of the network by NK modifiable connections. The neurons of the network are placed in a two dimensional space. Each neuron has neighbors in this space. Each neuron has lateral connections according to the core of convolution of the Mexican hat operating on its neighbors.

It is supposed that the weights of modifiable connections are initially random. For an input X vector the output of the network is a y vector with K components:

$$y_i = \sum_{j}^{n=1} W_{ij} X_j = X^T.W_i$$

In this equation W_i is the vector weight of neuron i, i.e. the vector with N components W_{ij}. According to the vector X and the initial configuration of the weights, there is a neuron i_0 whose output is the largest. This neuron is considered as the winner.

The activity of the neurons close to the neuron i_0 is facilitated, while that of distant neurons is inhibited. After balance, the output of the network reveals a zone of dominating activity around the winning neuron, surrounded by inactive zones, or slightly active. Modifiable connections are then adjusted according to the rule:

$$\Delta W_i = \alpha y_i (X - W_i)$$

The highest values of y_i can be found in the most active zone, i.e. around the neuron i_0, the corrections are important while they are very weak in the slightly active zones and null in the inactive zones. Qualitatively, the synaptic correction tends to make the neuron i_0 more selective to the X data. Indeed, for the neuron i_0 , the output is more important, one checks before the correction:

$$\forall i \neq i_0, y_{i0} = X^T W_{i0} > y_i = X^T W_i$$

After the correction, replacing $W_i \Longleftrightarrow W_i + \Delta W_i$ we obtain:

$$\begin{aligned} y_i &= X^T.(W_i + \Delta W_i) \\ &= X^T.[W_i + \alpha X^T W_i (X - W_i)] \\ &= X^T W_i (1 + \alpha X^T (X - W_i)) \end{aligned}$$

If the scalar product $X^T(X - W_i)$ is positive, after the correction of the weight, the selectivity of the neuron i_0 to the vector X is increased. The neurons linked to lateral connections, are also strongly concerned by the modification of the weights [Kohonen1].

8 The protocol

A neural network learns the signature (power consumption and electromagnetic analysis) of an instruction, and then recognize it later automatically.

We have to store hundreds traces for each instructions for a processor. So to be able to identify the instruction, we first set a pad to one, to change the power consumption, and then execute the instruction. To reduce the influence of the preceding instruction, we insert a "nop" instruction before the instruction we want to analyze. We used this instruction because we noticed the influence of the "nop" was none on the instruction just after. We keep the trace of the instruction (signature) and then repeat the procedure with the electromagnetic field.

So, at the end, we have hundreds signatures for each instruction. Some instructions are more complex than others (one or two parameters). We store several hundreds signatures. We just change one parameter (address of the instruction, data manipulated or address of the data manipulated) to be able to fix a large part of the signature. Once the total set of signatures per instruction is presented to the neural network, it is possible to class them. Each signature defines a zone in a space with its parameters, and the neural network determines the centroid of this area.

Then when you present the power signature or the electromagnetic signature of an instruction to the neural network, it is able to recognize it and to give you the class of the instruction.

9 A practical case: reversing a code

As we did for the correlation, we built two networks, one for the four clock cycles instructions and the second for the rest. Using same detector as before, with in input the shape of an average signature obtained for all 35 instructions, it is possible to isolate the patterns to be presented to the neural network. In the case of our processor, we succeeded in obtaining a reverse engineering of the code and the Hamming weights concerned in 93% of the cases. Figure 7 represents the measured trace with the values obtained at output of the neural network.

The same technique can be used to attack pin codes. Once a network has learned traces of wrong pin code comparison it is able to characterize the difference between the proposed pin code and the right one. We managed to defeat a pin code comparison on a old GSM phone card. It is also very interest-

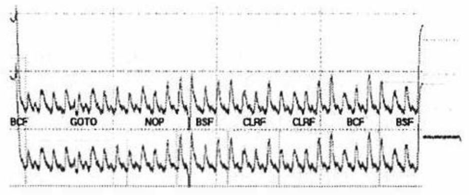


Figure 7: A power trace and the comments after code recognition

ing to notice that some PIN code comparison are still sensible to the timing attack. As the bytes are compared in a given order, and not a random one, it is possible to recover watching the answer time the PIN code. Of course, a very simple countermeasure is to randomize the comparison or to set a constant time before to answer.

In some smart cards, the processor manufacturer has hidden instructions to obtain a specific task. It is well known that some governmental agencies asked Digital to add/remove some instructions to the Alpha processor (Hamming weight). It is very difficult to recover these instructions. So with our method, when instructions appear in the power or EM traces, if they were never presented to the network before, the correlation between the form indicated by the network [Kohonen2] and the treated data shows that the network never met such an instruction in the past. The probability of a correct guess is low for all the outputs.

10 Work in progress

In the future we will focus on the signal processing part. Actually the acquisition chain (acquisition at 125 MHz 12 bits resolution) is good enough and we have to investigate new neural network to improve the selection of each instruction. We decided to start with neural networks based on the K-Nearest Neighbors. Anyway, it does not seem to be the unique solution and a Multi Layer Perceptron sounds quite nice too. Of course a QV based on Voronoi diagram can give results. But we have to test and select new criteria for the network (Manhattan distance). It may be possible to explain completely the influence of the pipeline using a source separator. The representation of the data is very important too, we have to avoid synchronization problems using a wavelet transform or modifying the architecture. We're actually testing a commercial crypto-processor with our network.

11 Conclusion

This article presents a use of traditional techniques of correlation and SOM to find the instructions executed by a very simple processor with only 35 instructions. In order to improve the signal noise ratio of the treated data, the electric field makes it possible under certain conditions to find the exact values of the handled data. Obviously it looks more like an academic case, than a directly usable attack. Indeed the processors for smart cards contain countermeasures, slowing down or preventing such attacks. However it is important to notice that the attacks on side channels will go on increasing, because the possibilities to correlate the data are multiple, and signal processing makes it possible to increase the effectiveness of the attacks.

12 Acknowledgments

We want to thank Cedric Volant for his help.

References

[Kocher] P. Kocher, J. Jaffe and B. Jun, *Differential Power Analysis*, In M. Wiener, editor, Advances in Cryptology - CRYPTO'99, vol. 1666 of Lecture Notes in Computer Science, pp. 388-397,Springer-Verlag, 1999. Also available at: http://www.cryptography.com/dpa/Dpa.pdf.

[Gandolfi] K. Gandolfi, C. Mourtel and F. Olivier, *Electromagnetic analysis : con-*

crete results, In Ko, Naccache, Paar editor, Cryptographic Hardware and Embedded Systems, vol 2162 of Lecture Notes in Computer Science, pp. 251-261, Springer-Verlag, 2001.

[Quisquater] J.-J Quisquater and D. Samyde, *ElectroMagnetic Analysis (EMA) Measures and Counter-Measures for Smart Cards*, in I. Attali and T. Jensen, editors, E-Smart Smartcard Programming and Security,, vol. 2140 of Lecture Notes in Computer Science, pp. 200-210, Springer-Verlag 2001.

[Boneh] D.Boneh, R.A. Demillo, and R. J. Lipton, *On the Importance of Checking Cryptographic Protocols for Faults*, in Proc. of Advances in Cryptology-Eurocrypt'97, Springer-Verlag, 1997, pp. 37-51.

[Anderson] R.Anderson, M.Kuhn, *Tamper resistance - A Cautionary Note*, Proc. of the Second USENIX Workshop on Electronic Commerce, USENIX Association, 1996.

[Messerges] T. Messerges and E .Dabbish, *Investigations of power analysis attacks on smartcards*, In Proc. of the USENIX Workshop on Smartcard Technology (Smartcard'99). USENIX Association, 1999.

[Fahn] P. N. Fahn and P. K. Pearson, *IPA : A new class of power attacks*, Proc. of CHES'99, editors C. K. Ko and C. Paar, Lecture Notes in Computer Science, vol. 1717, Springer-Verlag, pp. 173-186, 1999

[Kommerling] O. Kommerling and M. Kuhn, *Design principles for tamper-resistant smartcard processors*, In Proc. of the USENIX Workshop on Smartcard Technology (Smarcard'99), pp. 9-20. USENIX Association, 1999

[Coron] J-S. Coron, P. Kocher, and D. Naccache, *Statistics and Secret Leakage*, Financial Cryptography 2000 (FC'00), Lecture Notes in Computer Science, Springer-Verlag.

[Kuhn] M. Kuhn and R. Anderson, *Soft tempest: Hidden data transmission using electromagnetic emanations*, In D. Aucsmith, editor, Information Hiding, vol 1525 of Lecture Notes in Computer Science, pp 124-142. Springer-Verlag, 1998

[Biham] E. Biham, and A. Shamir, *Power Analysis of the Key Scheduling of the AES Canditates*, in Second Advanced Encryption Standard Canditate Conference, Rome, March 1999

[Kelsey] J. Kelsey, B. Schneier, D. Wagner, and C. Hall, *Side Channel Cryptanalysis of Product Ciphers*, in Proc. of ESORICS'98, Springer-Verlag, September 1998, pp. 97-110.

[Kohonen1] T. Kohonen, *Self-Organizing Maps*, Third Edition, in Information Science, Springer-Verlag, 2001.

[Kohonen2] T. Kohonen, *The self-organizing map*, Proceedings of the IEEE 78, pp. 1464-1480, 1990.

Breaking the Liardet-Smart Randomized Exponentiation Algorithm

Colin D. Walter

Comodo Research Laboratory
10 Hey Street, Bradford, BD7 1DQ, UK

colin.walter@comodo.net www.comodo.net

Abstract. In smartcard encryption and signature applications, randomised algorithms are used to increase tamper resistance against attacks based on side channel leakage. Recently several such algorithms have appeared which are suitable for RSA exponentiation and/or ECC point multiplication. We show that under certain apparently reasonable hypotheses about the countermeasures in place and the attacker's monitoring equipment, repeated use of the same secret key with the algorithm of Liardet and Smart is insecure against any side channel which leaks enough data to differentiate between the adds and doubles in a single scalar multiplication. Thus the scalar needs to be blinded in the standard way, or some other suitable counter-measures employed, if the algorithm is to be used safely in such a context.

Key words: *m*-ary exponentiation, Liardet-Smart randomized algorithm, ECC, addition chains, sliding windows, addition-subtraction chains, power analysis, SPA, SEMA, blinding, smartcard.

1 Introduction

Major progress in the theory and practice of side channel attacks [5, 6] on embedded cryptographic systems threatens to enable the capture of secret keys from *single* applications of cryptographic functions [10, 11, 14]. This is particularly true for the more computationally intensive functions such as exponentiation, which is a major process in many crypto-systems such as RSA, ECC and Diffie-Hellman.

Timing attacks on modular multiplication can usually be avoided easily by removing data-dependent conditional statements [16], but, with timing variations removed, attacks which make use of data-dependent variation in power and electro-magnetic radiation become easier. Initial attacks of this type required averaging over a number of exponentiations [8].

One counter-measure is to modify the exponent from e to $e+rg$ where r is a random number, typically 32-bits, and g is the order of the (multiplicative) group in which the exponentiation is performed [5]. This results in a different exponentiation being performed every time. However, if squares and multiplications can be distinguished during a single exponentiation, then use of the standard binary exponentiation algorithm immediately leads to exposure of the secret key.

For elliptic curve cryptography (ECC), the most efficient schemes for point addition and point doubling involve different numbers of operations in the field over which the curve is defined, and these numbers vary depending on the representation used for the curve. A counter-measure which reduces the likelihood of distinguishing between these point operations involves equalising the number and type of the component field operations [12] or making the point addition look exactly the same as two point doublings [1].

However, squares and multiplications in the field behave differently [13] and so there is no reason to believe that such recoding will necessarily hide fully the distinction between point additions and point doublings: for example, in [12], field squares appear for point additions, but field cubes when the same formula is used for point doublings. Side channels can distinguish these if the Hamming weight of arguments can be deduced. So exponentiation algorithms are chosen in which there is still an ambiguity in the correspondence between multiplications (i.e. point additions in ECC terms) and properties of the secret key (such as bit or digit values). *M*-ary exponentiation [4] for $m > 2$ provides one solution because each addition represents an unknown choice from a set of several non-zero digits.

If the same unblinded key value is re-used for many exponentiations, there is a danger that the repeated use of the same operand can be determined [14]. This would enable individual digits of the exponent base m to be identified and hence the key recovered. Unfortunately, particularly for ECC as opposed to RSA, applying the above exponent blinding technique is expensive when the secret key is typically only 192 bits. It adds about 17% to the cost of point multiplication. Hence randomised exponentiation algorithms may be a preferred option for ECC.

There are currently several algorithms which randomise the operations associated with specific inputs so that the exponentiation scheme is different on successive runs with the same data [7, 9, 17, 2, 3]. That of Liardet and Smart [7] uses a sliding window of random, variable width. If the attacker's equipment is insufficient to obtain information from a single EC point multiplication, then it seems that averaging over different multiplications with the same key would dilute any data dependency in the side channel leakage. However, we will show here that if individual point multiplications do leak information about what operation is being performed, then the secret key can be obtained straightforwardly. Indeed, one might even be better off with m-ary multiplication.

We begin by recalling the algorithm and looking at various parameters which might be chosen to improve efficiency or security. Next, the assumptions about the attacker's equipment and cryptosystem counter-measures are outlined. These are initially quite tight to make the presentation of the attack easier. The attack starts with extracting a least significant digit, and then uses this repeatedly to reconstruct one possible representation for the secret key. An essential part of the discussion is an assessment of the probability that the attack can be completed successfully. Before concluding with some counter-measures and alternatives, we explain how the attack can still be performed in a more realistic environment where the side channel leakage is much poorer.

2 The Algorithm

This section contains a brief outline of the (exponentiation) algorithm of Liardet and Smart. Because it generates an addition-subtraction chain rather than simply an addition chain, inverses have to be computed when it is applied. This means that applications to RSA cryptography are unlikely because of the expense of computing inverses. However, in elliptic curve cryptography (ECC), inverses are essentially for free. Hence, we will assume the algorithm is applied to an additive group, such as that formed by the points on an elliptic curve, and use appropriate terminology. Processing of the secret key k therefore produces a sequence of instructions which result in additions (A) and doublings (D) of group elements.

Suppose we wish to compute the element $Q = kP$ for a given positive integer k (the secret key) and a given member P of some group E. As in m-ary exponentiation, Liardet and Smart pre-compute the odd multiples iP of P for integers $i \in (-\frac{1}{2}m, \frac{1}{2}m]$ where $m = 2^R$, and then employ the standard sliding windows technique but with a window which has a random width showing up to R bits. In other words, k is recoded to obtain digits k_i ($0 \le i \le n$) which are determined using a randomly-chosen variable base m_i which divides m. The digits are chosen in the order $k_0, k_1, ..., k_n$ and the digit representation $k_n k_{n-1} ... k_1 k_0$ satisfies

$$k = ((...((k_n)m_{n-1}+k_{n-1})m_{n-2}+...)m_1+k_1)m_0+k_0 \qquad (1)$$

The group element multiplication processes these digits from most to least significant following the related scheme defined by

$$kP = m_0(m_1(...m_{n-2}(m_{n-1}(k_nP)+k_{n-1}P)+...)+k_1P)+k_0P \quad (2)$$

2.1 Code for the Key Recoding

More explicitly, if minmod is the function which returns a residue of minimal absolute value, the algorithm for choosing the digits is this:

RANDOMISED SIGNED m-ARY WINDOW DECOMPOSITION [7]:

```
i ← 0 ;
While k > 0 do
{   If (k mod 2) = 0 then
    {   m_i ← 2 ;
        k_i ← 0 ;
    }
    else
    {   Randomly choose base m_i ∈ {2^1,2^2,..., 2^R} ;
        k_i ← k minmod m_i ;
    } ;
    k ← (k-k_i)/m_i ;
    i ← i+1 ;
} ;
```

Here, both 1 and $\bar{1}$ could be allowed as digits for base 1, but that involves the added complication of a random bit to decide which to select, and also (to avoid non-termination) restricting the choice to only 1 when k reaches 1. Our attack would work also in these circumstances with few changes.

2.2 Efficiency Considerations

There are still some parameters to be chosen in the algorithm. Varying these affects efficiency, but there are also security implications. As we see later, certain choices will increase the difficulty of mounting the attack, forcing, in particular, more samples to be required.

The value for R has the greatest effect on efficiency. In elliptic curve applications, subtraction may have the same cost as addition. Then it will be unnecessary to store the negative pre-computed multiples of the input point. So only space for 2^R-2 multiples is likely to be required. Increasing R improves speed, but with diminishing returns for the space required for pre-computed values.

No suggestions are made in [7] about how to choose the m_i randomly. A uniform distribution is not very efficient, and indeed perhaps the least secure under this attack. It is most efficient to make the maximum possible use of the pre-computed values by choosing the maximum base size 2^R always. But, to maintain generality for later, suppose $m_i = 2^j$ is chosen with probability p_j when k is odd and $p_0 = 1$ is the probability of selecting base 2 when k is even.

Choosing $p_R = 1$ means that $m_i = 2^R$ whenever k is odd. This yields the usual m-ary sliding window method with fixed $m = 2^R$. Taking $R = 1$ yields the usual binary "square-and-multiply" algorithm. However, such choices would remove any non-determinism from the sequence of point operations.

Observe that biasing in the choice of m_i does not change the uniformity in the distribution of residues $k \bmod m_i$ inherited from k, assuming k is randomly chosen. This means that every new key value k generated during the recoding retains the same random properties: in particular, residues modulo 2^j will be uniform for every key encountered.

3 The Attack

3.1 Introduction & Initial Hypotheses

The purpose of randomised exponentiation algorithms is to frustrate side channel analysis by an attacker. In particular, they are counter-measures against using knowledge of the exponentiation process to extract the secret key k. Several different levels of leakage are possible, depending on the resources of the attacker. A poor signal-to-noise ratio (SNR) means that many samples have to be taken, and averaging the side channel leakage is one way of improving the SNR. So a critical parameter is whether or not the attacker's equipment is good enough for him to extract sufficient meaningful data from the side channel trace of a single scalar multiplication. If it is, then the standard key blinding described earlier suddenly fails to provide the data hiding protection afforded by averaging away local data dependencies. Improved equipment and laboratory techniques mean that this barrier might soon be breached without excessive expenditure [10, 11].

The categories of leakage which could be considered include the following:

i) individual point operations can be observed on power, EM or other side channel traces;
ii) point doublings and point additions can be distinguished from each other;
iii) re-use of operands can be observed; and
iv) operand addresses can be deduced.

Point (i) may hold simply because program instructions and data need to be fetched at the start of each point operation, and these cause different effects on the side channels than field operations. Point (ii) may then hold as a result of different patterns of field operations for point additions than for point multiplications. Properties (iii) and (iv) might hold as a result of being able to deduce Hamming weights of data and address words travelling along the bus.

Randomisation prevents the obvious averaging of the traces of many point multiplications which was used in initial power analysis attacks on the binary "square-and-multiply" algorithm. Here every point multiplication determines a different sequence of doublings and additions. With matched code for additions and doublings, averaging may hide the difference between the two operations because they are no longer separated in time, but in current implementations such averaging will certainly reveal the start and end of the individual point operations which make up the scalar multiplication.

The attack described here requires the SNR to be good enough to extract some useful data from single multiplications on the curve. Specifically, initially we assume that

- Adds and doublings can always be identified correctly and distinguished from each other using traces obtained from side channel leakage for a single point multiplication, and
- A number of traces are available corresponding to the same secret key value applied to different scalar multiplications.

Both of these hypotheses will be relaxed later to some extent, providing a more realistic scenario.

3.2 Overview of the Attack & Notation

The outline of the attack is as follows. For simplicity, by the first hypothesis,

- Every trace can be viewed as a word over the alphabet $\{A,D\}$.

Every occurrence of an A (add) in the trace splits the word into a prefix and a suffix which correspond to two integers k_p and k_s that are precisely defined in terms of k

and the position of A in the trace. All traces determine the same values to within ± 1. By looking at the patterns to the left of a given A, one obtains the residue of k_p modulo a small power of 2, and hence a few bits of k. Repeating this for the position of each A enables all the bits of k to be recovered.

Definition. *The **position** of a specific instance of character A or D in a trace word is the number of Ds which are to the right of the selected character.*

We will exploit a close relationship between positions in which A appears in traces and bits which are 1 in the corresponding position of the binary representation of k.

In order to be able to give examples, we fix the character order of words over $\{A,D\}$ to correspond to a left-to-right processing order. Thus, the digit sequence $1_2 0_2 \bar{1}_4 0_4$, with most significant bit on the left, is processed from most to least significant bit, i.e. from left to right, and so would result in the word $DADDDADD$. There are As in positions 2 and 5, and Ds in positions 0 to 5. In fact, every A is paired with a preceding D with the same position, and so one could view the DA combination as a single character. Then the position would correspond directly to a character index, counting from 0 at the right hand end, as in a binary representation.

The initial DA corresponds to the digit 1_2 and might be omitted if efficient initialisation takes place instead. Assuming this is the case, we will delete any initial Ds but leave the initial A as an unambiguous reminder that there is an initial digit to take into account. Thus words always commence with A. In the above example, $ADDDADD$ will be the word corresponding to the given digit sequence.

3.3 Properties of Key Digits

We now look at the sequence of digits generated by the Liardet-Smart algorithm. The notation used here is the same as for a fixed base, and many standard properties have analogues.

Lemma 1. *Suppose $k_n k_{n-1} \ldots k_1 k_0$ is a digit representation of k generated by the Liardet-Smart algorithm, with sequence $m_n, m_{n-1}, \ldots, m_1, m_0$ of bases. For some i, let $k_p^{(i)}$ denote the integer corresponding to the prefix $k_n k_{n-1} \ldots k_i$ and let $k_s^{(i)}$ denote the integer corresponding to the suffix $k_{i-1} \ldots k_1 k_0$. Then $k = k_p^{(i)} m^{(i)} + k_s^{(i)}$ where $m^{(i)} = \prod_{j=0}^{i-1} m_j$ and $|k_s^{(i)}| < m^{(i)}$.*

This lemma is obvious from the definition of the digit sequence given by equation (1) except, perhaps, for the last part. That part follows easily by induction. Digit k_i is chosen with $|k_i| \leq \frac{1}{2} m_i \leq m_i - 1$. This property for $i = 0$ starts the induction. Then the induction step is

$$|k_s^{(i+1)}| = |k_i m^{(i)} + k_s^{(i)}| < |(k_i + 1) m^{(i)}| \leq m_i m^{(i)} = m^{(i+1)}.$$

We will continue to use the notation $m^{(i)}$ for $\prod_{j=0}^{i-1} m_j$ and $k_p^{(i)}$ and $k_s^{(i)}$ for the key values associated respectively with the prefix $k_n k_{n-1} \ldots k_i$ and the suffix $k_{i-1} \ldots k_1 k_0$ of the digit sequence. The next lemma uses the equality and bound of Lemma 1 to identify two possible values for $k_s^{(i)}$, corresponding to it being a positive or negative residue of k modulo $m^{(i)}$:

Lemma 2. *With the previous notation, either*
i) $k_s^{(i)} = k \bmod m^{(i)}$ *and* $k_p^{(i)} = k \operatorname{div} m^{(i)}$, *or*
ii) $k_s^{(i)} = (k \bmod m^{(i)}) - m^{(i)}$ *and* $k_p^{(i)} = (k \operatorname{div} m^{(i)}) + 1$
where mod *returns the least non-negative residue, and* div *is integer division given by rounding down the real quotient.*

This shows that, whatever choices are made for the base elements, a given digit suffix can determine only one of two possible values when the product of the corresponding base elements is fixed. We would like at least the occurrence of the one which will make the corresponding prefix odd because it leads to an addition (A) which can be used to identify a corresponding point in the trace.

Lemma 3. *For all powers $2^j \leq k$, there is a choice of base elements $m_{i'}$ and an integer i such that $2^j = m^{(i)}$. On average, for at least $1-2^{-R}$ of all values of j, there are choices which make $k_p^{(i)}$ odd, and, for at least half of all values of j, there are choices which make $k_p^{(i)}$ even.*

Proof. The existence of the choice of basis elements is clear: taking $m_{i'} = 2$ for all i' allows one to satisfy the equality for $m^{(i)}$ with $i = j$. For that choice, k_i is the usual index i bit of k, and takes either parity with equal probability. It is the lowest bit of $k_p^{(i)}$, and so $k_p^{(i)}$ is the desired parity with probability $\frac{1}{2}$.

Other choices of base elements exist, and they may result in $k_p^{(i)}$ being odd even when the bit of interest in k is even. This increases the average number of cases for which oddness occurs. Instead of choosing $m_{j-1} = 2$, we try choosing $m_{j-i'} = 2^{i'}$ for any i' with $1 \leq i' \leq R$. The ability to select them depends on a corresponding bit of $k_p^{(j-R)}$ being 1 (otherwise base 2 must be chosen). These bits are independent and so the alternative bases can be chosen with probability ½. Each will give odd parity to the next k_p if chosen. Hence the prefix key corresponding to 2^j can be made odd with probability at least $1-2^{-R}$. ■

Of course, this argument just gives a lower bound on how many js will give rise to two key prefix and two suffix values. It doesn't guarantee that when both values are possible they will both appear with non-negligible frequencies. The actual relative frequencies appear to depend on the lowest R bits of the k_p corresponding to $m^{(t)} = 2^{j-R}$. However, in the next section, a lower bound on the ratio will be produced as necessary for each choice of these bits.

3.4 Recovering One Digit of *k*

In this section we show how to recover the least significant digit k_0 and associated base m_0 in one representation of k and how to identify the subset of traces which correspond to the associated prefix key k_p such that $k = k_p m_0 + k_0$. Exactly the same process yields other digits of k independently. Those digits can then be assembled together to give k in the manner described in the next section.

If Tr is the full set of all sample traces, then we denote by Tr_i the set of traces obtained by taking each member of Tr and deleting the suffix to the right of, but not including, the D of position i. Thus $Tr_0 = Tr$. Tr_i is partitioned into two complementary subsets: Tr_i^A which consists of those traces which terminate with A, and Tr_i^D which consists of those traces which terminate with D. We need to identify one of these subsets for each digit choice so that its neighbour to the left can be selected correctly. Tr_i^A always represents the odd choice for $k_p^{(i)}$, but some traces in Tr_i^D may contain only some of the operations for the rightmost prefix digit, and so not represent any $k_p^{(i)}$ properly.

The derivation here does make specific use of the fact that in this implementation $\bar{1}$ is not allowed as a digit for base 2. Similar arguments apply when $\bar{1}$ is allowed, but there is a duality which leaves a complicating ambiguity between the two values of $\pm k_s$ throughout the reconstruction process. This is only resolved when the complete value of k is reconstructed and under the assumption that the true sign of k is known.

Lemma 4. *Select any trace for key k. Then k is exactly divisible by 2^i where i is the uniquely defined integer such that AD^i is a suffix of the trace.*

Proof. Clearly, if k is divisible by 2^i then base 2 must be chosen for the lowest i digits, which are then all zeros. This leads to a character sequence D^i of i consecutive Ds as a suffix in every trace. If k is not divisible by 2^{i+1} then, whatever the next choice of base, the digit will be non-zero and hence cause A to be appended to the sequence, yielding the suffix AD^i. ■

This result enables these i occurrences of D to be identified with i least significant digits 0, each of base 2. Moreover, all traces confirm this conclusion. So, removing the digits one at a time,

Lemma 5. *If every trace in Tr has final character D then we may take $k_0 = 0$, $m_0 = 2$ and the traces of Tr_1 all represent the associated k_p.*

If k is odd, no digit has been deduced yet, and further work must be done.

Lemma 6. *Suppose* $k \equiv 1 \bmod 2^i$ *where* $i \leq R$. *Then* $k \equiv 2^i+1 \bmod 2^{i+1}$ *if Tr contains a trace with suffix* AD^iA. *If* $k \equiv 2^i+1 \bmod 2^{i+1}$ *then the probability that Tr contains no trace with suffix* AD^iA *is* $(1-p_i')^{|Tr|}$ *where* $p_i' = p_1+p_2+...+p_i$.

Proof. If $k \equiv 1 \bmod 2^{i+1}$ then a base of $m_0 = 2^{i+1}$ or larger will lead to suffix $D^{i+1}A$. However, a smaller base $m_0 = 2^j$ will lead to suffix D^jA with digit $k_0 = 1$ and the forced selection of base 2 at least $i+1-j$ times. This again leads to suffix $D^{i+1}A$.

Now suppose $k \equiv 2^i+1 \bmod 2^{i+1}$. A base of $m_0 = 2^{i+1}$ or larger again leads to suffix $D^{i+1}A$. However, the choice of base $m_0 = 2^i$ means lowest digit $k_0 = 1$ and next digit determined by k div 2^i, which is odd. Hence the suffix is AD^iA for that choice. Similarly, a base $m_0 = 2^j$ with $j < i$, will lead to suffix D^jA, digit $k_0 = 1$ and $k_p \equiv 2^{i-j} \bmod 2^{i-j+1}$. So this choice is followed by the forced selection of base 2 exactly $i-j$ times with associated digit 0. The subsequent digit is then odd, resulting in the overall suffix AD^iA.

Thus, suffix AD^iA guarantees $k \equiv 2^i+1 \bmod 2^{i+1}$ and it occurs precisely when the least significant base choice is 2^j with $j \leq i$. These choices occur for a given trace with probability $p_i' = p_1+p_2+...+p_i$. Hence suffix AD^iA will not happen for any trace in Tr with probability $(1-p_i')^{|Tr|}$. ■

Lemma 7. *Suppose* $k \equiv 1 \bmod 2^i$. *If Tr contains a trace with suffix* AD^iA *then* $k \equiv 2^i+1 \bmod 2^{i+1}$, *we may take* $k_0 = 1$ *and* $m_0 = 2^i$, *and the traces of* Tr_i^A *all represent the associated* k_p.

This lemma deals with the recognisable instances of $k \equiv 2^i+1 \bmod 2^{i+1}$. When base 2^j is chosen for any $j \leq i$, the suffix is AD^iA for these cases. As this occurs in p_i' of cases, so we expect Tr_i^A to contain approximately $p_i'|Tr|$ elements.

We will assume $k \equiv 1 \bmod 2^{i+1}$ if there is no suffix AD^iA but we know $k \equiv 1 \bmod 2^i$. By Lemma 6, this introduces a small probability of error which can be decreased by taking a larger sample if necessary, or by further analysis, such as through a more exhaustive analysis of suffixes and their expected frequencies than there is space for here. Note, however, that if $p_i' = 0$ then this choice of m_0 will not resolve which residue mod 2^{i+1} is correct. Hence an increase in security might be obtained by having $p_1 = p_2 = \ldots = p_i = 0$ where i is as large as possible.

Theorem 1. *Assume each base 2^i is selected with probability p_i for odd key values, and digit $\bar{1}$ is only used for bases greater than 2. Let $p_i' = p_1+p_2+\ldots+p_i$ and $\overline{p_i'} = 1-p_i'$. Suppose k is a random odd integer that has generated trace set Tr and j ($1 \leq j \leq R+1$) is such that Tr contains no trace with suffix AD^iA for any $i < j$. Then $k \equiv 1 \bmod 2^j$ with probability $\prod_{i=1}^{j-1}(1+\overline{p_i'}^{|Tr|})^{-1}$.*

Proof. We prove this by induction on j. For $j = 1$ the statement claims nothing, and so holds. For the induction step, assume the statement holds for some $j \leq R$. Suppose also that Tr contains no trace with suffix AD^iA for any $i \leq j$. By the induction hypothesis, $k \equiv 1 \bmod 2^j$ with probability $\prod_{i=1}^{j-1}(1+\overline{p_i'}^{|Tr|})^{-1}$.

Since k is random, the two possibilities for $k \bmod 2^{j+1}$ are equally likely. So, by Lemma 6, no occurrence of suffix AD^jA means $k \equiv 1 \bmod 2^{j+1}$ with probability $(1+\overline{p_j'}^{|Tr|})^{-1}$. This factor just needs multiplying into the product to obtain the claim for j+1 in place of j. ■

Theorem 2. *With assumptions and notation as in Theorem 1, suppose k is odd and j is minimal such that $1 \leq j \leq R+1$ and Tr contains a trace with suffix AD^jA. Then $k \equiv 2^j+1 \bmod 2^{j+1}$ with probability*

$$\prod_{i=1}^{j-1}(1+\overline{p_i'}^{|Tr|})^{-1}.$$

If $j \leq R$ we may take $k_0 = 1$ and $m_0 = 2^j$ and then the set of traces for the associated k_p is Tr_j^A.

Proof. Theorem 1 shows that, for the given definition of j, $k \equiv 1 \bmod 2^j$ with probability $\prod_{i=1}^{j-1}(1+\overline{p_i'}^{|Tr|})^{-1}$. If $k \equiv 1 \bmod 2^{j+1}$ then, as in the proof of Lemma 6, all traces must terminate with suffix $D^{j+1}A$, which is not the case. Hence $k \equiv 2^j+1 \bmod 2^{j+1}$ with the stated probability.

For the base $m_0 = 2^j$, $k \equiv 1 \bmod m_0$ and so the associated digit is $k_0 = 1$. However, $k_p = k \text{ div } 2^j$ is odd, which forces the next digit to be non-zero. Hence A is the next operation leftwards after the suffix D^jA which corresponds to m_0. Thus, the relevant traces for the next digit are those of Tr_j^A. ■

The values of k for which no least significant digit has yet been assigned are those satisfying $k \equiv 1 \bmod 2^{R+1}$. Picking maximal base $m_0 = 2^R$ gives $k_0 = 1$ and makes k_p even. The associated set of prefix traces should be Tr_R^D. A possible difficulty with this definition is that for some traces removing the suffix D^RA may split subsequences which correspond to one digit. However, every choice of base 2^i corresponds to a suffix D^iA where $i \leq R$, and must be followed by a number of instances of base 2 with digit 0 which makes the total modular division by at least 2^{R+1}. Hence the suffix D^RA corresponds to the operations for a whole number of digits. Therefore Tr_R^D does indeed contain traces which represent only operations for sequences of complete digits, and so those traces all represent the same key value.

Theorem 3. *With the same assumptions and notation as in Theorem 1, suppose every trace in Tr has suffix $D^{R+1}A$. Then, with probability $\prod_{i=1}^{R}(1+\overline{p_i'}^{|Tr|})^{-1}$, $k \equiv 1 \bmod 2^{R+1}$ and we may pick $m_0 = 2^R$. For this choice $k_0 = 1$, k_p is the common key for Tr_R^D, k_p is even, and $Tr_R^D = Tr_R$ has the same cardinality as Tr.*

3.5 Combining Digits to Recover *k*

For every position j at which there is an occurrence of A in some trace of Tr, the procedures of the previous subsection can be applied to Tr_j^A to obtain a base and digit at that point. These digits are used when determining a digit sequence for k. Starting at $j = 0$, the digits are selected iteratively. As well as a digit and base, each trace set Tr_j^A gives rise to another trace set defined at some position $j' > j$. We will show that:

- For this definition of j', the next digit is determined by whichever is appropriate of $Tr_{j'}^A$ or $(Tr_{j'}^A)_R^D$.

Here we need to check on the definition of the trace subsets. If applied iteratively, the procedures above would actually determine smaller and smaller subsets: each time we apparently take a subset of the traces from the previous step. However, because only two key values (one odd, one even) are associated with any position, every prefix which represents the operations of a complete number of digits must correspond to the odd key if it terminates with A and the even key if it terminates with D. Of course, every trace prefix terminating with A must consist of the operations for a whole number of digits since A cannot appear in the middle of the sequence of operations for a single digit. So every trace in Tr_j^A is generated from a key value which is common to them all. Hence, the full set Tr_j^A can be used to determine the next digit, not just the subset of Tr_j^A determined by the procedures above.

In the case of the prefix trace set Tr_{j+R}^D, it is not clear which traces are generated by a complete key. In some cases, the final D may not be the final operation of the digit sequence from which it was derived. Hence, the subset $(Tr_j^A)_R^D$ must be used, not Tr_{j+R}^D. However, the construction observed that every such trace had suffix $D^{R+1}A$. So $(Tr_j^A)_R^D$ has the same cardinality as Tr_j^A. Hence the trace subsets bulletted above are indeed the correct ones to use for the key digits, and they do not progressively decrease in size.

The process of digit determination only begins to fail once a leading instance of A is encountered: Theorem 2 guarantees progress up to that point. Traces are not all the same length. Some will use a large base for the most significant digit. Their initial Ds are deleted, giving them fewer instances of D overall, making their traces shorter. These traces are simply discarded when fully processed. The procedures above still apply to the subset. Again, following Theorem 2, further digits can still be defined until the trace set becomes empty. However, once the first (i.e. shortest) traces run out, the remaining key is representable by a single digit, so it is bounded in absolute value by $2^{R-1}-1$. Each increment of the position in the trace set reduces the representable key by a factor of 2. Eventually, assuming there are enough traces, the initial A of the longest trace has a digit bounded by 1, and so must be 1. Hence k is completely determined. "Enough" traces would be present if, for example, base 2 were chosen for the most significant digit. Insufficient traces just increases the number of possible values of k which may need testing by a small factor (under 2^{R-1}).

3.6 The Probability of Error

We have been careful to obtain the probability of error in each digit in order i) to see if it is feasible to recover the key and ii) to see how the probabilities p_i might be adjusted in the algorithm definition to provide improved security.

The procedures of §3.4 define the probabilities in terms of the size of the trace set being employed at that time. Generally, it is equal to the cardinality of a set of the form Tr_j^A. This is equal to $|Tr|$ times the number of DAs in position j divided by the number of DAs or Ds in that position. This can be approximated by $|Tr|$ times the overall probability p_A of DA divided by the overall probability p_D of DA or D. Since the choice of base $m = 2^i$ produces $i-1$ occurrences of D followed by one of DA when $i > 0$, $|Tr_j^A| \approx \pi\,|Tr|$ where

$$\pi = \frac{p_A}{p_D} = \frac{p_1+p_2+\ldots+p_R}{p_0+p_1+2p_2+\ldots+Rp_R}$$

For a uniform distribution this works out at $\pi = \frac{2}{R+3}$ where typically we might expect $R = 3$; and for 2^R-ary sliding windows it works out at $\pi = \frac{1}{R+1}$.

In fact, the formula under-estimates the average size of Tr_j^A. Some positions do not have any occurrences of A, and we do not use the associated trace subsets. This increases the average for those positions which do have occurrences of A.

Next, the distribution of base choices in the reconstructed key differs from that generated by the re-coding process. Suppose k is odd for the set of traces at some point during the reconstruction. In Theorem 2, the distribution of odd residues $k \bmod 2^{R+1}$ is uniform. So, neglecting the assumed small numbers of incorrectly assigned cases resulting from some of the possible suffixes not occurring, base 2^j will be selected for the reconstructed key with probability 2^{-j} for $0 < j \le R$ and produce an odd next key. Further, base 2^R will turn up in the remaining 2^{-R} cases of odd keys but produce an even next key. In half of all cases, an even key will lead to an even key. Consequently, out of every 2^R+2 digit choices in the reconstruction, on average 2^R will be odd and 2 will be even.

By Theorems 2 and 3, the probability of the reconstructed key being correct is a product of factors of the form $(1+\overline{p_i}^{|Tr_j^A|})^{-1}$. These factors can be approximated by $(1+\overline{p_i}^{\pi|Tr|})^{-1}$. Since choosing base 2^j leads to j such factors, there is essentially one such factor for every bit of k. The exceptions are where an even key causes base 2 and digit 0. Then there is no doubt about the correctness of the digit 0. This last case occurs for $2(2^R+2)^{-1}\log_2 k$ bits of the initial key k. Otherwise, for odd keys, the relative frequency of different bases means that the factor $(1+\overline{p_i}^{\pi|Tr|})^{-1}$ will appear on average for $2^{R-i}(2^R+2)^{-1}\log_2 k$ bits if $0 < i < R$ and for $2(2^R+2)^{-1}\log_2 k$ bits if $i = R$.

Because $p_R' = p_1+p_2+...+p_R = 1$, the factor for $i = R$ is 1, and so can be ignored. Hence,

Lemma 8. *The key k can be recovered with a probability approximately*

$$\prod_{i=1}^{R-1}(1+\overline{p_i}'^{\pi|Tr|})^{-2^{-i}n}$$

where $n = (1+2^{1-R})^{-1}\log_2 k$ *and* π *is as defined above.*

The property $\overline{p_i}' \leq 1-p_1$ provides a lower bound for this product. Consequently,

Lemma 9. *For a uniform distribution of base, the key k can be recovered with a probability at least*

$$(1+(\tfrac{R-1}{R})^{\pi|Tr|})^{-n}$$

where $n = (1-2^{1-R})(1+2^{1-R})^{-1}\log_2 k$ *and* $\pi = \frac{2}{R+3}$.

For specific choices, it is possible to evaluate the product in Lemma 8 exactly. A typical choice would be to have a key with 192 bits, $R = 3$ (which requires storing two pre-computed multiples, namely P and $3P$), and a uniform choice of base, i.e. $p_1 = p_2 = p_3 = ⅓$. Then $\pi = ⅓$ and the product in Lemma 8 is just under 2^{-31} for $|Tr| = 9$. So, if a key can be reconstructed and checked for correctness in unit time,

Theorem 4. *If doubles and adds can be distinguished on individual traces, and traces are captured from 9 applications of the same unblinded 192-bit key, then the Liardet-Smart algorithm with uniform selection of base $\leq 2^3$ can be broken with a computational effort of about $O(2^{31})$. With twice as many traces, the computational effort falls to under $O(2^{10})$.*

Of course, the full force of all the patterns available and their relative frequencies has not yet been applied. Hence the danger is probably substantially under-estimated. Once a possible key has been recovered, there is considerable unused data in the traces that has not yet been used and can be investigated for checking purposes. In the uniform case, about $\frac{R+1}{R+3}$ of the data is so far unused – that in the complementary sets Tr_j^D. This contains information about digits whose bases were not aligned with those of the reconstructed representation of k. Choosing a different base from that of the reconstruction process described above will provide confirmation about the correctness of each bit of k. Indeed, each trace has to be consistent with some choice of bases, and the rightmost inconsistency in a trace will usually be very close to the rightmost bit in error. There is insufficient space here to improve the probabilities which are a consequence of this approach, but the computational feasibility of the attack is already assured.

If the attacker is unable to distinguish clearly between adds and doubles, then the unused data vastly increases his ability to make corrections. Moreover, as each digit is obtained through a purely local extraction of data from traces, it is easy to automate an exhaustive process to check for the overall best digit solutions using all traces, and hence prioritise the order for considering the most likely values for k. However, for the data that has been used, any indistinctness between A and D is unimportant. In this attack, it is only necessary to establish whether or not an A has appeared at each position. The relative frequency of As means that the certainty of this can be determined with high degree just by increasing the number of traces sufficiently.

4 Counter-Measures

Our formulae for bounding the accuracy repeatedly used the probabilities of smaller bases much more than larger bases, and the accuracy improves when these probabilities are increased at the expense of the probabilities of larger bases. This is consistent with the greater ambiguity afforded by digits of larger bases. Thus we recommend not using a uniform choice for the base, but employing a strong bias towards large bases, such as was illustrated in §2.2. In the extreme, the standard, non-randomised, m-ary exponentiation technique is obtained, and this is not susceptible to the attack.

The cost of key masking is not entirely trivial in the context of ECC. Adding a 32-bit random multiple of the group order to the key increases the point multiplication cost by some 17% for 192-bit keys, although it is a much smaller fraction of the total encryption cost. Adding a smaller random multiple is probably ineffective if it results in a number of repetitions of the same key value within the lifetime of the key. The highly repetitive nature of the traces resulting from the same prefix keys turning up again and again means that a duplicated key could be assumed if, and only if, traces matched closely enough.

The "double-and-add-always" method of computation provides a good measure of protection, but is expensive. The attacker then has to determine whether or not the result of the addition is used before he can mount the attack. This is much more difficult than distinguishing the two operations. Hence traces will be susceptible to much more frequent errors, and a much greater number of traces will have to be recovered.

There are alternative randomised algorithms for which this type of attack does not apply, and others that display similar weaknesses. That of Oswald and Aigner [9] can be attacked in a similar way. MIST [15, 17] does not exhibit the same repetition of key values during key processing, and so may be a safer choice. A new algorithm by Itoh *et al.* [18] may also be worthy of consideration.

5 Conclusion

It might have been hoped that the Liardet-Smart algorithm would avoid the cost of any additional counter-measures such as key blinding when the same secret key is repeatedly re-used, but this now appears not to be so. Specifically, the key needs to be masked, or the pattern of adds and doubles has to be well hidden for individual point multiplications.

Of course, there are many circumstances in which the algorithm is clearly of value, such as ECDSA, for which a different random key is used every time. Then, for suitable parameter choices, the space of keys generating a given pattern of adds and doubles is infeasibly large, and so cannot be attacked successfully without additional data.

References

[1] C. H. Gebotys & R. J. Gebotys, *Secure Elliptic Curve Implementations: An Analysis of Resistance to Power Attacks in a DSP Processor*, Cryptographic Hardware and Embedded Systems – CHES 2002, B. Kaliski, Ç. Koç & C. Paar (editors), Lecture Notes in Computer Science, **2523**, Springer-Verlag, 2002, *to appear*.

[2] J. C. Ha & S. J. Moon, *Randomized Signed-Scalar Multiplication of ECC to Resist Power Attacks*, Cryptographic Hardware and Embedded Systems – CHES 2002, B. Kaliski, Ç. Koç & C. Paar (editors), Lecture Notes in Computer Science, **2523**, Springer-Verlag, 2002, *to appear*.

[3] K. Itoh, J. Yajima, M. Takenaka & N. Torii, *DPA Countermeasures by Improving the Window Method*, Cryptographic Hardware and Embedded Systems – CHES 2002, B. Kaliski, Ç. Koç & C. Paar (editors), Lecture Notes in Computer Science, **2523**, Springer-Verlag, 2002, *to appear*.

[4] D. E. Knuth, *The Art of Computer Programming*, vol. 2, "Seminumerical Algorithms", 2nd Edition, Addison-Wesley, 1981, 441–466.

[5] P. Kocher, *Timing Attack on Implementations of Diffie-Hellman, RSA, DSS, and other systems*, Advances in Cryptology – CRYPTO '96, N. Koblitz (editor), Lecture Notes in Computer Science, **1109**, Springer-Verlag, 1996, 104–113.

[6] P. Kocher, J. Jaffe & B. Jun, *Differential Power Analysis*, Advances in Cryptology – CRYPTO '99, M. Wiener (editor), Lecture Notes in Computer Science, **1666**, Springer-Verlag, 1999, 388–397.

[7] P.-Y. Liardet & N. P. Smart, *Preventing SPA/DPA in ECC Systems using the Jacobi Form*, Cryptographic Hardware and Embedded Systems – CHES 2001, Ç. Koç, D. Naccache & C. Paar (editors), Lecture Notes in Computer Science, **2162**, Springer-Verlag, 2001, 391–401.

[8] T. S. Messerges, E. A. Dabbish & R. H. Sloan, *Power Analysis Attacks of Modular Exponentiation in Smartcards*, Cryptographic Hardware and Embedded Systems (Proc CHES 99), C. Paar & Ç. Koç (editors), Lecture Notes in Computer Science, **1717**, Springer-Verlag, 1999, 144–157.

[9] E. Oswald & M. Aigner, *Randomized Addition-Subtraction Chains as a Countermeasure against Power Attacks*, Cryptographic Hardware and Embedded Systems – CHES 2001, Ç. Koç, D. Naccache & C. Paar (editors), Lecture Notes in Computer Science, **2162**, Springer-Verlag, 2001, 39–50.

[10] J.-J. Quisquater & D. Samyde, *ElectroMagnetic Analysis (EMA): Measures and Counter-Measures for Smart Cards*, Smart Card Programming and Security (E-smart 2001), Lecture Notes in Computer Science, **2140**, Springer-Verlag, 2001, 200–210.

[11] J.-J. Quisquater & D. Samyde, *Eddy current for Magnetic Analysis with Active Sensor*, Smart Card Programming and Security (E-smart 2002), Lecture Notes in Computer Science, Springer-Verlag, 2002, *to appear*.

[12] M. Joye & J.-J. Quisquater, *Hessian Elliptic Curves and Side Channel Attacks*, Cryptographic Hardware and Embedded Systems – CHES 2001, Ç. Koç, D. Naccache & C. Paar (editors), Lecture Notes in Computer Science, **2162**, Springer-Verlag, 2001, 402–410.

[13] C. D. Walter & S. Thompson, *Distinguishing Exponent Digits by Observing Modular Subtractions*, Topics in Cryptology – CT-RSA 2001, D. Naccache (editor), Lecture Notes in Computer Science, **2020**, Springer-Verlag, 2001, 192–207.

[14] C. D. Walter, *Sliding Windows succumbs to Big Mac Attack*, Cryptographic Hardware and Embedded Systems – CHES 2001, Ç. Koç, D. Naccache & C. Paar (editors), Lecture Notes in Computer Science, **2162**, Springer-Verlag, 2001, 286–299.

[15] C. D. Walter, *Improvements in, and relating to, Cryptographic Methods and Apparatus*, UK Patent Application 0126317.7, Comodo Research Laboratory, 2001.

[16] C. D. Walter, *Precise Bounds for Montgomery Modular Multiplication and Some Potentially Insecure RSA Moduli*, Proceedings of CT-RSA 2002, Lecture Notes in Computer Science, **2271**, Springer-Verlag, 2002, 30–39.

[17] C. D. Walter, *MIST: An Efficient, Randomized Exponentiation Algorithm for Resisting Power Analysis*, Proceedings of CT-RSA 2002, Lecture Notes in Computer Science, **2271**, Springer-Verlag, 2002, 53–66.

[18] K. Itoh, J. Yajima, M. Takenaka & N. Torii, *DPA Countermeasures by Improving the Window Method*, Cryptographic Hardware and Embedded Systems – CHES 2002, B. Kaliski, Ç. Koç & C. Paar (editors), Lecture Notes in Computer Science, **2523**, Springer-Verlag, 2002, *to appear*.

A Protected Division Algorithm

Marc Joye and Karine Villegas
Gemplus Card International, Card Security Group, France
{marc.joye, karine.villegas}@gemplus.com, http://www.gemplus.com/smart/

Abstract

Side-channel analysis is a powerful tool for retrieving secrets embedded in cryptographic devices such as smart cards. Although several practical solutions have been proposed to prevent the leakage of sensitive data, mainly the protection of the basic cryptographic operation itself has been thoroughly investigated. For example, for exponentiation-based cryptosystems (including RSA, DH or DSA), various exponentiation algorithms protected against side-channel analysis are known. However, the exponentiation algorithm itself or the underlying crypto-algorithm often involve division operations (for computing a quotient or a remainder). The first case appears in the normalization (resp. denormalization) process in fast exponentiation algorithms and the second case appears in the data processing before (resp. after) the call to the exponentiation operation.

This paper proposes an efficient division algorithm protected against simple side-channel analysis. The proposed algorithm applies equally well to software and hardware implementations. Furthermore, it does not impact the running time nor the memory requirements.

Keywords. Division algorithms, smart cards, side-channel analysis, SPA protected implementations.

1 Introduction

Significant progress has been made these last years to secure cryptographic devices (e.g., smart cards) against side-channel analysis. Side-channel analysis [2, 3] is a clever technique exploiting side-channel information (e.g., power consumption) to retrieve secret information involved in the execution of a carelessly implemented crypto-algorithm. The threat is now clearly understood by implementors and various countermeasures have been suggested.

The basic operation underlying most public-key crypto-algorithms is the modular exponentiation. To name a few, this includes the RSA cryptosystem, the Diffie-Hellman key exchange or the DSA signature scheme. The resistance of modular exponentiation with respect to side-channel analysis is discussed in many papers (e.g., see [4] where both attacks and counter-measures are presented). A far less studied operation is that of division: to the authors' best knowledge, there is no paper in the public literature addressing this issue. This is most unfortunate as nearly all implementations of exponentiation-based cryptosystems use the division operation as well.

Several specialized modular multiplication algorithms (and therefore the corresponding modular exponentiation algorithms) require a normalization step involving an integer division. Typical examples include Barrett algorithm [5] or Quisquater algorithm [6] (see also [7]). For computing $a \cdot b \bmod m$, these two algorithms take on input a normalization factor of the form $\mu = \lfloor 2^t/m \rfloor$. If the division algorithm used for evaluating μ is prone to side-channel analysis then the value of m (or some related information) can be recovered. When m is a secret data, this compromises the security of the cryptosystem. For example, this occurs when RSA decryption (or signature) is speeded up through Chinese remaindering [8] because then modulus m is successively one of the two secret RSA primes, p_1 and p_2. A second example of division algorithm manipulating secret data is when RSA is used with Chinese remaindering and operand x in the computation of $x^d \bmod \{p_1, p_2\}$ is first explicitly reduced modulo p_i prior to the exponentiation $x^d \bmod p_i$, for $i = 1, 2$.

An algorithm commonly used for computing integer divisions is the classical binary pencil-and-paper method (or a variant thereof). This algorithm presents the advantage of requiring no extra memory requirements. However, as we will see, it may yield the value of quotient $q = a \operatorname{div} b$ during its computation by simple side-channel analysis. This paper is aimed at transforming this algorithm into a division algorithm protected against simple side-channel analysis while preserving the efficiency (memory-wise) of the classical algorithm. Actually, the resulting algorithm will not only be protected against simple side-channel analysis but will further be faster than the classical algorithm, with the same memory requirements. As a result, we obtain a protected division algorithm that is few greedy in memory and is particularly suited to a hardware implementation or to a software implementation in a constrained environment like a smart card.

The rest of this paper is organized as follows. The next section reviews the classical binary pencil-and-paper division algorithm. Its security towards simple side-channel analysis is studied in Section 3. Building on the pencil-and-paper method, we then propose in Section 4 our protected yet more efficient division algorithm. Finally, we conclude in Section 5.

Disclaimer. This paper only addresses security against *simple* side-channel analysis, that is, side-channel analysis from a *single* measurement of certain side-channel information. In particular, it is not concerned with differential analysis (such as DPA) or more sophisticated methods.

2 Pencil-and-Paper Division Method

Given a and b on input, the binary pencil-and-paper algorithm evaluates the quotient $q = a \operatorname{div} b$ (alternatively, we use the notation $q = \lfloor \frac{a}{b} \rfloor$) and the remainder $r = a \bmod b$. The binary representations of a and b are respectively given by $a = (a_{m-1}, \ldots, a_0)_2$ and $b = (b_{n-1}, \ldots, b_0)_2$ with $b_{n-1} \neq 0$.

It is easy to see that the pencil-and-paper division of two integers amounts to the simpler problem of dividing a $(n+1)$-bit integer A by a n-bit integer b and then to re-iterate the process [1]. We must have $0 \leq A/b < 2$, which is satisfied whenever $b_{n-1} \neq 0$ (see above restriction).

Since we are working in basis 2, the two possible values for the quotient bit $\lfloor A/b \rfloor$ are 0 or 1. So we subtract b from A and test whether the obtained result is nonnegative; if so then $\lfloor A/b \rfloor = 1$, and $\lfloor A/b \rfloor = 0$ otherwise. Remark that $\lfloor A/b \rfloor = 1$ if and only if $A - b \geq 0$.

The length of $r = a \bmod b$ plus the length of $q = a \operatorname{div} b$ is smaller than or equal to $(m+1)$ bits. Indeed, the length of r is at most n bits since $r < b$; and the length of q is at most $(m - n + 1)$ bits since $q = \lfloor a/b \rfloor \leq \lfloor a/(b_{n-1}2^{n-1}) \rfloor = a \operatorname{div} 2^{n-1} = (a_{m-1}, \ldots, a_{n-1})_2$, a $(m - n + 1)$-bit value. In order to save memory, the quotient and remainder will be written in the register containing a (augmented with one leading bit).

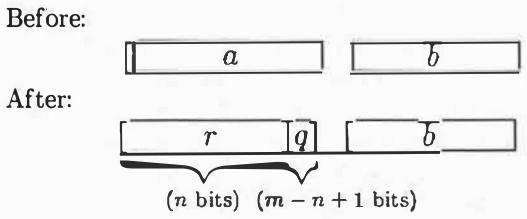


Figure 1: Memory configuration.

It is useful to introduce some notations. For a k-bit integer a, we denote by $\mathtt{SHL}_k(a, 1)$ the operation consisting in shifting a of one bit to the left; the outgoing bit is affected to the carry. For two k-bit integers a and b, we write $\mathtt{ADD}_k(a, b)$ for the addition of a and b; variable carry is set to 1 if there is a carry in the addition and carry is set to 0 otherwise. Remark that $\mathtt{SHL}_k(a, 1)$ can equivalently be obtained as $\mathtt{ADD}_k(a, a)$. There is usually no subtraction operation available for large integers. The subtraction of b from a is obtained by first computing the two's complement of b, denoted by $\bar{b}$, and then by adding $\bar{b}$ to a. Indeed, if b is a k-bit integer then $b + \bar{b} = 2^k$ and so $a - b = a + \bar{b} \pmod{2^k}$. We write $\mathtt{CPL2}_k(a)$ the operation consisting in taking the two's complement of a k-bit integer a. Symbols $\vee$, $\wedge$ and $\oplus$ refer to the bit-wise logical operations OR, AND and XOR, respectively. For a bit σ, the negation of σ (i.e., its complementary value) is denoted by $\neg\sigma$. Finally, the notation $\operatorname{lsb}(a)$ refers to the least significant bit of an integer a.

We can now present the classical binary pencil-and-paper division algorithm. On input a and b, this algorithm computes both the values of $a \operatorname{div} b$ and of $a \bmod b$. In order to work, a is artificially augmented with one bit (initially set to 0) at the most significant position. Moreover, to ease the exposition, variable A represents the n most significant bits of a, i.e., $A = (0, a_{m-1}, \ldots, a_{m-n+1})$.

```
Input:  $a = (0, a_{m-1}, \ldots, a_0)_2$
        $b = (b_{n-1}, \ldots, b_0)_2$
Output: $q = a \operatorname{div} b$ and $r = a \bmod b$
$b \leftarrow$ CPL2$_n(b)$ /* $b$ = "$-b$" */
for $j = 1$ to $(m - n + 1)$ do
    $a \leftarrow$ SHL$_{m+1}(a, 1)$ /* Shift */
    $\sigma \leftarrow$ carry
    $A \leftarrow$ ADD$_n(A, b)$ /* Subtract */
    $\sigma \leftarrow \sigma \vee$ carry
    if $(\neg\sigma)$ then /* Correction */
                    $b \leftarrow$ CPL2$_n(b)$
                    $A \leftarrow$ ADD$_n(A, b)$
                    $b \leftarrow$ CPL2$_n(b)$
            else lsb$(a) = 1$
endfor
$b \leftarrow$ CPL2$_n(b)$
```

Figure 2: Binary pencil-and-paper division algorithm.

Remainder r is in A followed by the quotient q. The correctness of the algorithm follows by observing that if $a \leftarrow$ SHL$_{m+1}(a, 1)$ generates a carry then $a_m = 1$ (before shifting) and so b must be subtracted; moreover if $a_m = 0$ (before shifting) and $A \leftarrow$ ADD$_n(A, b)$ generates a carry (i.e., $A - b \geq 0$ before the subtraction) then again b must be subtracted.

Example 1 *Suppose we want to compute* $a \operatorname{div} b$ *and/or* $a \bmod b$ *with* $a = 4096 = (1000000000000)_2$ *and* $b = 81 = (1010001)_2$. *The 2's complement of* b *is* $\bar{b} = (0101111)_2$. *We obtain* $r = 46 = (101110)_2$ *and* $q = 50 = (110010)_2$.

								a							σ
	0	1	0	0	0	0	0	0	0	0	0	0	0	0	$\star$
Shift	1	0	0	0	0	0	0	0	0	0	0	0	0	$\underline{0}$	0
Subtract	1	1	0	1	1	1	1								0
Correction	1	0	0	0	0	0	0								
Shift	0	0	0	0	0	0	0	0	0	0	0	0	$\underline{0}$	0	1
Subtract	0	1	0	1	1	1	1							1	1
Shift	1	0	1	1	1	1	0	0	0	0	0	$\underline{0}$	1	0	0
Subtract	0	0	0	1	1	0	1							1	1
Shift	0	0	1	1	0	1	0	0	0	0	$\underline{0}$	1	1	0	0
Subtract	1	0	0	1	0	0	1								0
Correction	0	0	1	1	0	1	0								
Shift	0	1	1	0	1	0	0	0	0	$\underline{0}$	1	1	0	0	0
Subtract	1	1	0	0	0	1	1								0
Correction	0	1	1	0	1	0	0								
Shift	1	1	0	1	0	0	0	0	$\underline{0}$	1	1	0	0	0	0
Subtract	0	0	1	0	1	1	1							1	1
Shift	0	1	0	1	1	1	0	$\underline{0}$	1	1	0	0	1	0	0
Subtract	1	0	1	1	1	0	1								0
Correction	0	1	0	1	1	1	0	$\underline{0}$	1	1	0	0	1	0	

3 Security Analysis

This section explains why security may be a concern when implementing a division algorithm.

Back to the binary pencil-and-paper division algorithm (Fig. 2), we see that the quotient is constructed bit by bit. According to its value, each quotient bit is obtained by a different sequence of operations. At step j in the for-loop, if the obtained quotient bit, say q_j, has value 0 then the following operations were performed:

- a shift [SHL$_{m+1}(a, 1)$]
- an addition [ADD$_n(A, b)$]
- a "correction" [CPL2$_n(b)$; ADD$_n(A, b)$; CPL2$_n(b)$]

along with some logical operations; whereas if $q_j = 1$ then the following two operations were performed:

- a shift [SHL$_{m+1}(a, 1)$]
- an addition [ADD$_n(A, b)$]

along with some logical operations. If it is possible to distinguish these two sequence of operations during the course of the algorithm then the value of quotient bit q_j (and thus of the whole quotient q) can be recovered. Such a means may be provided by monitoring the power consumption (i.e., the side-channel is the power consumption). The next figure represents a power trace resulting from the execution of the pencil-and-paper division algorithm on a chip equipped with a crypto-coprocessor. The operands are large numbers and the various operations (shift, addition or two's complement) are performed by the crypto-coprocessor.

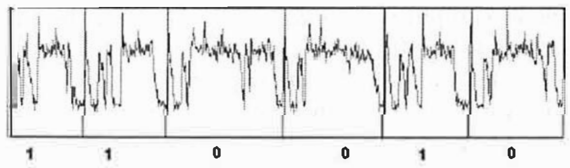


Figure 3: A power trace of the pencil-and-paper division algorithm.

We can identify two different patterns in Fig. 3: one corresponds to the case $q_j = 0$ (i.e., a longer pattern involving an additional "correction") and the other one corresponds to the case $q_j = 1$. Therefore, the value of quotient $q = a \operatorname{div} b$ can easily read from the power trace.

If quotient q (or related data) is secret, this above implementation is not secure. Consider the example of RSA implemented with Chinese remaindering. Let $N = p_1 p_2$ be an RSA modulus (the values of p_1 and p_2 are secret). The computation of $y = x^d \bmod N$ is carried out as $y = \mathrm{CRT}(y_1, y_2)$ where $y_i = x_i{}^d \bmod p_i$ with $x_i = x \bmod p_i$, for $i = 1, 2$. Suppose that x_1 and x_2 are computed by the binary pencil-and-paper algorithm as given in Fig. 2. Then, by simple power analysis[1] (SPA), the values of $q_1 := \lfloor x/p_1 \rfloor$ and $q_2 := \lfloor x/p_2 \rfloor$ can be recovered from the corresponding power traces. Suppose further that $x = N - r$ for some $0 < r < p_1$. Then

$$q_1 = \left\lfloor \frac{N-r}{p_1} \right\rfloor = \left\lfloor p_2 - \frac{r}{p_1} \right\rfloor = p_2 - 1$$

and so the secret RSA primes are given by $p_2 = q_1 + 1$ and $p_1 = N/p_2$.

4 A Protected Method

The previous analysis illustrates that non-constant code may reveal sensitive data, thereby compromising the security of the cryptosystem. A first idea to make the code constant consists in adding some dummy operations and in making implicit the `if-then-else` statement. Such a solution is however unsatisfactory as dummy operations penalize the running time. This is especially true when the dummy operations are time-consuming

Rather, we exploit the following observation. At each iteration, the pencil-and-paper algorithm (Fig. 2) computes $a \leftarrow 2a - b\,2^{m-n+1}$. In the case of an unsuccessful guess (i.e., when $\sigma = 0$), one has to restore the value of a by setting $A \leftarrow A + b$. This restoring step can be avoided by noting that $2(a + b\,2^\alpha) - b\,2^\alpha = 2a + b\,2^\alpha$. We then obtain the non-restoring variant of the classical binary pencil-and-paper division algorithm. An additional variable, σ', keeps track of the value of bit σ in the previous iteration. Bit β keeps track of the 'sign' of b.

[1] That is, a side-channel analysis using a single power consumption measurement as side-channel information.

```
Input:  $a = (a_{m-1}, \dots, a_0)_2$
        $b = (b_{n-1}, \dots, b_0)_2$
Output: $q = a \operatorname{div} b$ and $r = a \bmod b$
$\sigma' \leftarrow 1$; $\beta \leftarrow 1$
for $j = 1$ to $(m - n + 1)$ do
  $a \leftarrow \mathrm{SHL}_{m+1}(a, 1)$; $\sigma \leftarrow$ carry
  if ($\sigma'$)
    then if ($\beta$) then $b \leftarrow \mathrm{CPL2}_n(b)$; $\beta \leftarrow 0$
         $A \leftarrow \mathrm{ADD}_n(A, b)$; $\sigma \leftarrow \sigma \vee$ carry
    else if ($\neg\beta$) then $b \leftarrow \mathrm{CPL2}_n(b)$; $\beta \leftarrow 1$
         $A \leftarrow \mathrm{ADD}_n(A, b)$; $\sigma \leftarrow \sigma \wedge$ carry
  if ($\sigma$) then $\mathrm{lsb}(a) = 1$
  $\sigma' \leftarrow \sigma$
endfor
if ($\neg\beta$) then $b \leftarrow \mathrm{CPL2}_n(b)$
if ($\neg\sigma$) then $A \leftarrow \mathrm{ADD}_n(A, b)$
```

Figure 4: Non-restoring binary division algorithm.[3]

The previous algorithm does not behave regularly and so may also be subject to side-channel analysis. According to the values of β and of σ', register b is unchanged or replaced by its two's complement. A first step towards side-channel protection consists in always performing a two's complement followed by an addition, whatever the values of β and σ'. To this end, when register b needs not to be replaced by its two's complement, a dummy two's complement —on a register, say register c, that does not impact the computation— is executed. We call d_{addr} the address of the register containing the value that will be replaced by its two's complement (d_{addr} will be b_{addr} or c_{addr}).

It is also worth noting that σ can be updated as $\sigma \leftarrow \sigma'(\sigma \vee \texttt{carry}) + (\neg\sigma')(\sigma \wedge \texttt{carry})$, which can equivalently be rewritten as

$$\sigma \leftarrow (\sigma \wedge \sigma') \oplus (\sigma \wedge \texttt{carry}) \oplus (\sigma' \wedge \texttt{carry}) \;.$$

Finally, noting that the shift operation sets the least significant bit of a to 0, the line [`if` (σ) `then` $\mathrm{lsb}(a) = 1$] can be replaced by $\mathrm{lsb}(a) \leftarrow \sigma$.

We use β and γ variables to keep track of the 'sign' of the value contained in the registers located at b_{addr} and c_{addr}, respectively. The convention is $\beta = 0$ (resp. $\gamma = 0$) when the value located at b_{addr} (resp. c_{addr}) is the original value, and $\beta = 1$ (resp. $\gamma = 1$) when the value located at b_{addr} (resp. c_{addr}) is the two's complement of the original value. We

[3] Again variable A represents the n most significant bits of variable a, i.e., $A = (0, a_{m-1}, \dots, a_{m-n+1})$.

have the following truth table:

σ'	β	γ	β	γ
0	0	0	1	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	0	1

where-from we derive the outgoing values $\beta \leftarrow \neg\sigma'$ and $\gamma \leftarrow \gamma \oplus \sigma' \oplus \beta$.

Putting all together, we finally obtain the following algorithm.

```
Input:  $a = (a_{m-1}, \ldots, a_0)_2$
        $b = (b_{n-1}, \ldots, b_0)_2$
Output: $q = a \operatorname{div} b$ and $r = a \bmod b$
$\sigma' \leftarrow 1$; $\beta \leftarrow 1$; $\gamma \leftarrow 1$
for $j = 1$ to $(m - n + 1)$ do
    $a \leftarrow \mathrm{SHL}_{m+1}(a, 1)$ /* Shift */
    $\sigma \leftarrow$ carry; $\delta \leftarrow \sigma' \oplus \beta$
    $d_{addr} \leftarrow b_{addr} + \delta(c_{addr} - b_{addr})$
    $d \leftarrow \mathrm{CPL2}_n(d)$ /* Two's complement */
    $A \leftarrow \mathrm{ADD}_n(A, b)$ /* Addition */
    $\sigma \leftarrow (\sigma \wedge \sigma') \oplus (\sigma \wedge \text{carry}) \oplus (\sigma' \wedge \text{carry})$
    $\beta \leftarrow \neg\sigma'$; $\gamma \leftarrow \gamma \oplus \delta$; $\sigma' \leftarrow \sigma$
    $\mathrm{lsb}(a) = \sigma$
endfor
/* Final correction */
if $(\neg\beta)$ then $b \leftarrow \mathrm{CPL2}_n(b)$
if $(\neg\gamma)$ then $c \leftarrow \mathrm{CPL2}_n(c)$
if $(\neg\sigma)$ then $A \leftarrow \mathrm{ADD}_n(A, b)$
```

Figure 5: Our protected division algorithm.[4]

One may argue that our algorithm is not code-constant because of the three last `if-then` statements. We note however that the two first are not mandatory to make the algorithm working but are merely performed to reset the registers containing b and c to their original values. Finally, the last `if-then` only reveals the least significant bit of the quotient; when this is a secret value a dummy operation can be applied to mask the potential addition $\mathrm{ADD}_n(A, b)$.

[4]As in Figs. 2 and 4, variable A represents the n most significant bits of a (i.e., $A = (0, a_{m-1}, \ldots, a_{m-n+1})$).

Example 2 *We take the same example as before: $a = 4096 = (1000000000000)_2$ and $b = 81 = (1010001)_2$ ($\bar{b} = 0101111$). As detailed below, we obtain $r = 46 = (101110)_2$ and $q = 50 = (110010)_2$.*

	a														σ	β
	0	1	0	0	0	0	0	0	0	0	0	0	0	0	$\star$	1
Shift	1	0	0	0	0	0	0	0	0	0	0	0	0	$\underline{0}$	0	1
$\mathrm{CPL2}_n(b)$																
$\mathrm{ADD}_n(A,b)$	1	1	0	1	1	1	1								0	0
Shift	1	0	1	1	1	1	0	0	0	0	0	0	$\underline{0}$	0	1	0
$\mathrm{CPL2}_n(b)$																
$\mathrm{ADD}_n(A,b)$	0	1	0	1	1	1	1							1	1	1
Shift	1	0	1	1	1	1	0	0	0	0	0	$\underline{0}$	1	0	0	1
$\mathrm{CPL2}_n(b)$																
$\mathrm{ADD}_n(A,b)$	0	0	0	1	1	0	1							1	1	0
Shift	0	0	1	1	0	1	0	0	0	0	$\underline{0}$	1	1	0	0	0
$\mathrm{CPL2}_n(c)$																
$\mathrm{ADD}_n(A,b)$	1	0	0	1	0	0	1								0	0
Shift	0	0	1	0	0	1	0	0	0	$\underline{0}$	1	1	0	0	1	0
$\mathrm{CPL2}_n(b)$																
$\mathrm{ADD}_n(A,b)$	1	1	0	0	0	1	1								0	1
Shift	1	0	0	0	1	1	0	0	$\underline{0}$	1	1	0	0	0	1	1
$\mathrm{CPL2}_n(c)$																
$\mathrm{ADD}_n(A,b)$	0	0	1	0	1	1	1							1	1	1
Shift	0	1	0	1	1	1	0	$\underline{0}$	1	1	0	0	1	0	0	1
$\mathrm{CPL2}_n(b)$																
$\mathrm{ADD}_n(A,b)$	1	0	1	1	1	0	1								0	0
$\mathrm{CPL2}_n(b)$	(Final corr. on b)															
Final corr.	0	1	0	1	1	1	0	$\underline{0}$	1	1	0	0	1	0	0	0

The corresponding power trace is given in Fig. 6. Remark that the power trace is now the repetition of a same pattern, regardless the value of the quotient bit.

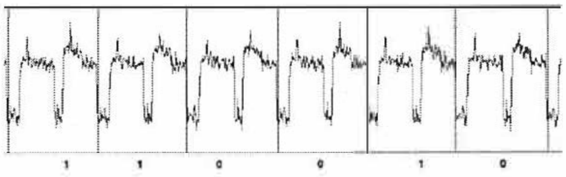


Figure 6: A power trace of our division algorithm.

Finally, as a side-effect, it is easy to see that, on average, our protected algorithm outperforms the classical pencil-and-paper method, with the same memory constraints (cf. Fig. 1).

Table 1: Comparison with the classical method.

	ADD_n	SHL_{m+1}	$\mathrm{CPL2}_n$
Classical	$\frac{3}{2}(m - n + 1)$	$m - n + 1$	$m - n + 3$
Protected	$m - n + \frac{3}{2}$	$m - n + 1$	$m - n + 2$

5 Conclusions

This paper presented a new division algorithm preventing simple side-channel analysis. The proposed algorithm is well suited to a hardware implementation or to a software implementation in a constrained environment. Remarkably, it does not require additional resources (time or memory) and is even faster than the classical binary method.

Obviously, we note that SPA-like analysis highly depends on the hardware and special care must be paid by the implementor. In this respect, the proposed method can be seen as a useful framework for designing protected and, as shown in the paper, efficient division algorithms.

References

[1] D.E. Knuth, *The Art of Computer Programming – Seminumerical Algorithms*, Addison-Wesley, 2000.

[2] P. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems," in *Advances in Cryptology – CRYPTO '96*, vol. 1109 of *Lecture Notes in Computer Science*, pp. 104–113, Springer-Verlag, 1996.

[3] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Advances in Cryptology – CRYPTO '99*, vol. 1666 of *Lecture Notes in Computer Science*, pp. 388–397, Springer-Verlag, 1999.

[4] T.S. Messerges, E.A. Dabbish, and R.H. Sloan, "Power analysis attacks of modular exponentiation in smartcards," in *Cryptographic Hardware and Embedded Systems (CHES '99)*, vol. 1717 of *Lecture Notes in Computer Science*, pp. 144–157, Springer-Verlag, 1999.

[5] P. Barrett, "Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processing," in *Advances in Cryptology – CRYPTO '86*, vol. 263 of *Lecture Notes in Computer Science*, pp. 311–323, Springer-Verlag, 1987.

[6] J.-J. Quisquater, "Encoding system according to the so-called RSA method, by means of a microcontroller and arrangement implementing this system." U.S. patent #5,166,978, Nov. 1992.

[7] A. Bosselaers, R. Govaerts, and J. Vandewalle, "Comparison of three modular reduction functions," in *Advances in Cryptology – CRYPTO '93*, vol. 773 of *Lecture Notes in Computer Science*, pp. 175–186, Springer-Verlag, 1994.

[8] J.-J. Quisquater and C. Couvreur, "Fast decipherment algorithm for RSA public-key cryptosystem," *Electronics Letters*, vol. 18, pp. 905–907, Oct. 1982.

A Java Reference Model of Transacted Memory for Smart Cards

Erik Poll
University of Nijmegen, the Netherlands
Pieter Hartel
University of Twente, the Netherlands
Eduard de Jong
Sun Microsystems, Inc., Santa Clara, USA

Abstract

Transacted Memory offers persistence, undoability and auditing. We present a Java/JML Reference Model of the Transacted Memory system on the basis of our earlier separate Z model and C implementation. We conclude that Java/JML combines the advantages of a high level specification in the JML part (based on our Z model), with a detailed implementation in the Java part (based on our C implementation).

1 Introduction

In a previous paper [6] we introduced Transacted Memory as an efficient means to implement atomic updates of arbitrarily sized information on smart cards. Smart cards need such a facility, as a transaction can be aborted by a *card tear*, i.e. by pulling the smart card out of the Card Acceptance Device (CAD), at any moment. A patent application has been filed for this Transacted Memory [5]. Its design allows a much smaller implementation overhead than the transaction mechanism in the current Java Card API[1], which does not even provide generational, logging, or multiple concurrent transactions.

[1] Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. or other countries, and are used under license.

In our earlier paper we provided a succinct abstract Z specification [13] of the system, a first Z refinement that takes into account the peculiarities of EEPROM memory (i.e. byte read versus block write), a second Z refinement that deals with card tear, and, finally, an (inefficient) C implementation. (The inefficiency is due to the use of many simple for-loops that search the memory; we are working on a VHDL specification of a hardware module that will replace the for-loops by efficient parallel searches but this is beyond the scope of the present paper.) The C implementation has been coded in such a way that it also serves as a SPIN [8] model.

From our earlier work we concluded that a formal connection between specification and implementation would have been highly desirable, yet such a connection cannot be obtained using Z and C. While a formal connection can be established using SPIN, we believe the readability leaves much to be desired, as specification and implementation tend to be intertwined in a SPIN model.

In the present paper we adopt an integrated approach to specification and implementation that solves the problems of readability and the lack of a formal connection between specification and implementation. We use the Java/JML [9] modelling method and tools, which means we write formal specifications by annotating the Java code with invariants, preconditions, and postconditions, using the specification language JML (see `www.jmlspecs.org`). These formal specifications can then be compiled into runtime-checks [4], providing a convenient way of checking specifications against code. The Java/JML modelling method and the runtime assertion checker ensure a strong, formal connection between Java implementation and JML specification.

In the present work we apply Java/JML to what we hope will become a component of a future version of the Java Card technology. JML has already been used to specify the entire Java Card API [11, 12], and other tools than the runtime assertion checker have already been used to verify JML specifications of Java Card applets [3, 1].

The contributions of the present paper are:

- Several bugs have been detected and repaired in the implementation of the Transacted Memory.
- We make the pre- and postconditions of the memory operations explicit in the JML specifications. The readability of these specifications is better because the reader does not have to trawl through the entire Z specification to discover the pre- and postconditions. The connection between specification and implementation is formal, and has been checked using the runtime assertion checker.
- The previous C implementation cum SPIN model relied on implicit methods of modelling the recovery from card tears. In the Java/JML model we use exception handling as an explicit, clearer method for modelling recovery. This allows us to test the behaviour of the Java implementation in the presence of (simulated) card tears, and to use JML to precisely specify the conditions that should hold after a card tear.
- We contribute a *reference model* of the Transacted Memory system to SUN's collection, instead of just a *reference implementation*. The difference is in the presence of the formal JML specification.

In Section 2 we review briefly how Transacted Memory works. Section 3 describes the Java implementation of the system, Section 4 discusses the JML specification for this Java implementation. The last section concludes.

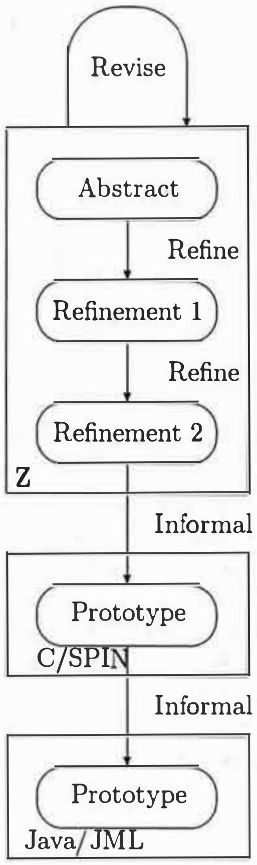


Figure 1: The process

2 The Transacted Memory

Figure 1 describes the relationship between the various specifications and implementations of the Transacted Memory system. The Java/JML reference model, which is the subject of this paper, was derived by hand from the closely corresponding C implementation cum SPIN model for the Java part, and from the final refinement of the Z specification for the JML part. While Java and C are similar in many ways, there are some important differences, discussed in Section 3 below. Here we concentrate on how Transacted Memory works, giving excerpts of the abstract Z specification to make the present paper self contained; the details are in [6, 2].

Transacted Memory is designed around two notions: tags and information sequences. A *Tag* is merely a unique address, i.e. an identifier of a particular information sequence. An information sequence is a sequence of *Info*'s, where *Info* is the unit of data stored and retrieved. A sequence of *Info*'s would be used to store a collection of object instances that are logically part of a transaction.

The abstract Z specification (below) makes no specific assumptions about either component:

$[Tag, Info]$

The existence of a finite set of available tags is assumed (*tags*), as well as limits on the size of the memory (*msize*). There may be several generations of the information associated with a tag, and there is a maximum number of generations that may be associated with any tag (*maxgen*):

$$\begin{array}{|l} tags : \mathbb{F}\, Tag \\ msize : \mathbb{N}_1 \\ maxgen : \mathbb{N}_1 \end{array}$$

The abstract Z specification represents the memory system as two partial functions *assoc* and *size* and a set *committed*, as shown below. We have omitted the constraints on the partial functions and the set:

$$\begin{array}{|l} \_\_ AMemSys \_\_\_\_\_\_ \\ assoc : tags \rightarrowtail \mathrm{seq}(\mathrm{seq}\, Info) \\ size : tags \rightarrowtail \mathbb{N}_1 \\ committed : \mathbb{P}\, tags \\ \hline \ldots \\ \hline \end{array}$$

The *assoc* function associates a tag with a sequence of sequences of information. The first sequence of information represents the current information associated with a tag. Any further information sequences give older generations of this information, in order of increasing age.

The *size* function gives the length of the information sequences associated with a tag. The *committed* set records those tags for which the current state of the transacted data has been committed.

Operations are provided to write a new generation, and to read the current or older generations. All generations associated with a tag have the same size, although this could be generalised.

The transaction processing capability of the memory is supported by a commit operation, which makes the most recently written information the current generation. The oldest generation is automatically purged should the number of generations for a tag exceed a preset maximum. It should be noted that the support for recording multiple generations, which can be useful for logging, essentially comes for free, i.e. without any additional implementation cost.

As an example, the abstract Z specification of the operation *ACommit* is shown below. The operation commits the current generation of information associated with a tag. The tag must have an associated information sequence, which is flagged as committed.

$$\begin{array}{|l} \_\_ ACommit \_\_\_\_\_\_ \\ \Delta AMemSys \\ t? : tags \\ \hline t? \in \mathrm{dom}\, assoc \\ assoc\ t? \neq \langle\rangle \\ committed' = committed \cup \{t?\} \\ \hline \end{array}$$

The Transacted Memory must be used in such a way that a sequence of operations either completes normally, or that a sequence is interrupted at an arbitrary moment by a card tear. A recovery operation `Tidy` is provided to return the Transacted memory to a known state. The idea is that each time the card is inserted in the CAD, the recovery operation is automatically started.

Transacted Memory thus provides undoability (by being able to revert to a previous generation) and persistence (by using EEPROM technology). These are precisely the ingredients necessary to support transactions [10].

To provide this functionality, Transacted Memory maintains a certain amount of book-keeping information. In its most abstract form, the bookkeeping information records three items:

- The length of the information sequence that is associated with a tag.
- The different generations of information associated with each tag. It is possible that there is no information associated with a tag.
- Which tags are currently committed.

The details of the Z specification may be found in a technical report [2]; here we focus on the API of the Transacted Memory, taken from our previous paper [6] and shown in Figure 2, because this is where the pre- and postconditions of the Java/JML specification provide the major contribution to readability and rigour.

3 A Java implementation of Transacted Memory

The Java implementation was obtained by manually transliterating the C code to Java code. This is not difficult as the languages are close, and for a program of this size (1200 lines) the effort involved is small. We have been careful in transliterating the C code, and we are confident that our Java implementation closely mimics the C implementation. There are two essential differences between the Java implementation and the C implementation, as explained below.

Static Type Checking

The C implementation contains several macros to define "types" for the different kinds of numeric values (bytes) that are used, such as generations, locations, page numbers, tags, versions, etc.:

```
#define Gen  byte  /* 0 .. maxgen   */
#define Loc  byte  /* 0 .. msize-1 */
#define PageNo  byte
#define Tag  byte  /* 0 .. tsize-1 */
#define Ver  byte  /* 0 .. 2 */
#define Inf  byte  /* 0 .. isize-1 */
#define Seq  byte  /* 0 .. ssize    */
```

These are just macros, and although they increase the readability of the code, they do not provide any type-safety.

In the Java implementation we have chosen to use different classes for these different kinds of values. This is inefficient since we make what is just a simple byte into an object. The inefficiency is not a primary concern here; we believe it to be more important for a reference model to be as clear and concise as possible[2]. Modelling bytes by classes has the advantage of providing type-safety, as for instance 'generations' and 'tags' are no longer assignment-compatible. Interestingly, this increased type safety immediately revealed a bug in the C code (and SPIN model): in one place a 'version number' was used in a place where a 'page number' was expected. This bug seems to have been a simple typo in the C code. This bug was not discovered in the model checking using SPIN, nor in testing of the C implementation, because the test harness for the Transacted Memory used there was fairly restricted.

The discovery of this bug illustrates the value of a statically enforced type system. Especially for code like that of the Transacted Memory, which is littered with different 'kinds' of bytes, it is easy to confuse a byte representing a page number with a byte representing a 'version'. It is a pity that C and Java do not have type-safe enumeration types, and that JML does not improve the level of expressiveness of the Java/JML combination in this respect.

[2]Also, the Java Card technology offers the possibility to optimize API components, such as the transacted memory API, in the offcard converter.

`typedef struct { Gen old, new ; byte cnt ; } GenGenbyte ;`	structure used to hold the number of the oldest and newest generation, and the number of generations.
`typedef struct { Size size ; Info data[ssize] ; } InfoSeq ;`	structure used to hold an information sequence and its size.
`GenGenbyte DGeneration( Tag ) ;`	Return all available information for the given tag. The result is undefined if the tag is not in use.
`Tag DNewTag( Size ) ;`	Return an unused tag of the specified size. The result is undefined if no tag is available.
`void DTidy( ) ;`	Recover from an interrupted write operation.
`InfoSeq DReadGeneration( Tag, Gen ) ;`	Read the information sequence of a given tag and generation. The information sequence is undefined if the tag is not in use.
`InfoSeq DRead( Tag ) ;`	Read the information sequence of the current generation associated with the given tag.
`void DCommit( Tag ) ;`	Commit the current generation for the given tag. The operation has no effect if the tag is already committed.
`void DRelease( Tag ) ;`	Release all information associated with the given tag. The operation has no effect if the tag is not in use.
`void DWriteFirst( Tag, InfoSeq ) ;`	Write to a tag immediately after the `DNewTag` operation. The result is undefined if insufficient space is available.
`void DWriteUncommitted( Tag, InfoSeq ) ;`	Write to a tag whose current generation is uncommitted.
`void DWriteCommittedAddGen( Tag, InfoSeq ) ;`	Write to a tag whose current generation has been committed, and whose maximum number of generations has *not* been reached.
`void DWriteCommittedMaxGen( Tag, InfoSeq ) ;`	Write to a tag whose current generation has been committed, and whose maximum number of generations *has* been written. The oldest generation will be dropped.

Table 1: Transacted Memory data structures and functions for C.

Modelling card tear

The second, and more important, aspect in which the Java implementation essentially differs from the C implementation is that we use Java's exception mechanism to model card tears. We introduce a special exception class `CardTearException`, and a card tear is simulated by throwing this exception. This is useful, because it allows us

1. to *test* the behaviour of the program when card tears occur; in the Java method that models atomic writes to EEPROM we can easily simulate random card tears by randomly choosing to throw a `CardTearException` or not, before or after the atomic write to EEPROM.
2. to *specify* in JML the properties that should hold after a card tear occurs; this will be discussed in Section 4.

In fact, though we will not pursue this point in this paper, a card tear can be modelled very accurately as an (uncatchable) Java exception, for which the power-on mechanism of the card provides the exception handler; see [7].

In a later stage we will also introduce Java exceptions to signal that there is insufficient free transacted memory to carry out an operation, as discussed at the end of Section 4.

4 JML specifications for the Java implementation

The Java Modeling Language (JML) [9] is a behavioural interface specification language tailored to Java. JML is developed primarily by Gary T. Leavens at Iowa State University. Java programs can be specified using JML by annotating them with invariants, pre- and postconditions, and other kinds of assertions. JML combines features of Eiffel (or 'Design by Contract') and model-based approaches, such as Larch/LSL and VDM.

JML annotations are written as a special kind of Java comments. This means they are ignored by normal Java compilers, but can be used by special tools for JML. The tools we have used on our JML-annotated code are the JML type-checker and the JML runtime assertion compiler [4]. Both these tools can be downloaded from `www.jmlspecs.org`. The runtime assertion compiler turns annotations into runtime checks, so that any violation of an annotation at runtime produces an error.

To create the JML specifications for the Java implementation, elements of the Z specifications and of the informal comments given in the C code were converted into pre- and postconditions, class invariants, and loop invariants. The JML specifications we have written are partial in the sense that they do not give a complete specification of Transacted Memory. Still, the specifications do express the main properties that should hold for the Transacted Memory, and have proven to be sufficiently detailed to find bugs, as we will discuss later.

Figure 2 gives an example of a JML specification, namely the specification of the method `DWriteUncommitted`. The JML specification is written between the annotation markers `/*@` and `@*/`.

The first three lines of the JML specification, starting with `requires`, give the *precondition* of the method. Here the precondition is that the `tag` should be in use, the information sequence `i` should be of the right length, and the `tag` should not be committed. When doing runtime assertion checking, any invocation of `DWriteUncommitted` which violates these preconditions will produce an error message[3].

The next two lines, starting with `ensures`, give the *postcondition* of the method. The first of these lines says that if we read back the value for `tag` using `DRead` we get the value `i` we just assigned to it, the second says that the `tag` is still not committed. When doing runtime assertion checking, any invocation of `DWriteUncommitted` which does not

[3]Actually, JML is so expressive that some JML assertions are not decidable, e.g. assertions using the keyword `forall` to quantify over an infinite domain; these (parts of) JML assertions are not compiled into runtime checks.

```
/* Write to a tag whose current generation is uncommitted.  */

/*@ requires ddata[tag.value].tagInUse;       // tag in use
    requires ddata[tag.value].size == i.seq; // i of right length
    requires ! ddata[tag.value].committed;   // tag uncommitted

     ensures DRead(tag).equals(i);           // i written successfully
     ensures ! ddata[tag.value].committed;   // tag still uncommitted

     signals (CardTearException) ! ddata[tag.value].committed;
     signals (CardTearException) DRead(tag).equals(i)
                             || DRead(tag).equals(\old(DRead(tag)))
 @*/
 public void DWriteUncommitted(Tag tag, InfoSeq i)
                             throws CardTearException;
```

Figure 2: JML specification of DWriteUncommitted

establish these postconditions will produce an error message.

Finally, the last lines of the JML specification, starting with signals, give the *exceptional postcondition*. Whereas ensures clauses specify the 'normal' postconditions, i.e. properties that should hold after normal termination of a method invocation, signals clauses specify properties that should hold at the end of a method invocation if an exception is thrown. The first signals clause here says that if a CardTearException is thrown then the tag remains uncommitted. The second signals clause says that if a CardTearException is thrown, then either

```
DRead(tag).equals(i)
```

or

```
DRead(tag).equals(\old( DRead(tag)))
```

i.e. reading back the value for tag either produces the 'new' value i just written or it produces the 'old' value of DRead(tag). The JML keyword \old is used here to refer to the value an expression had before execution of the method.

Note that the information sequence i may consist of several bytes, and that a single DWriteUncommitted operation may require several writes to EEPROM. EEPROM is typically written block by block, where the block size depends on the particular EEPROM. So the second signals clause states the atomicity of the DWriteUncommitted operation!

When doing runtime assertion checking, any invocation of DWriteUncommitted which throws a CardTearException and which does not establish the exceptional postconditions will produce an error message. Throwing an exception that is not a CardTearException will also produce an error message, as there are no signals clauses allowing other exceptions to be thrown.

Everything the runtime assertion checker does could be programmed by hand, as tests in the code – the C implementation has a number of these tests scattered through the code –, but note that for something like the second signals clause above this is far from trivial! It would involve catching and re-throwing exceptions at the end of the method, as well as somehow recording the 'old' value that DRead(tag) has in the pre-state. The JML runtime assertion tool compiles all this into the code automatically, which is useful, as it means we can concentrate on the essentials.

The other three write-operations – DWriteFirst, DWriteCommittedAddGen, and DWriteCommittedMaxGen – have specifications very similar to the one discussed above. The only difference is in their preconditions.

The specification of DWriteUncommitted

above is still incomplete. For example, it does not specify that the older generations of the tag are left unchanged. Still, specifications like this turn out to be detailed enough to give useful feedback when checking them at runtime. As discussed below, several problems with the implementation came to light when performing runtime assertion checks.

Bug 1 – Uncommitting new generations

Performing tests of the Transacted Memory the runtime assertion checker immediately reported that `DWriteCommittedAddGen` and `DWriteCommittedMaxGen` do not establish their postconditions; more specifically, they fail to establish

```
ensures !ddata[tag.value].committed;
```

The implementations of these methods forget to reset the committed flag of the tag. This bug was not discovered using SPIN, because the test harness used there committed every new generation immediately after the write operation.

Note that even in the Java/JML model we could have forgotten this postcondition, and then we would not have discovered the problem either. However, by systematically writing specifications for all the operations we believe one is less likely to forget something like this.

Bug 2 - Inadvertent commit

After Bug 1 was repaired, a second bug was discovered by runtime assertion checking. We also repaired the SPIN model and re-ran the model checker on that, and found the same error there.

The operations `DWriteCommittedAddGen` and `DWriteCommittedMaxGen` start a new, uncommitted, generation, but a card tear at a certain point in their execution may inadvertently commit the new generation written. Both `DWriteCommittedAddGen` and `DWrite-CommittedMaxGen` first write the data for the new generation. This may take several atomic writes, but the last of these implicitly records that the whole write has been successful (in effect, making the whole writing of the data atomic). Then the commit flag is cleared – also atomically, but separate from the last write for the data. If a card tear occurs immediately after the data is written, but before the commit flag is cleared, the tag will appear committed to the recovery process, whereas in reality it should be uncommitted. The recovery process was not designed to detect this, and indeed a warning to this effect appears in the original Z specification [2, page 34].

The solution which we have implemented is to use not a boolean commit flag, but a three-valued flag, so that a `DWriteCommitted-AddGen` or `DWriteCommittedMaxGen` interrupted at the precise point above can be detected during recovery. (An alternative solution would be to store the last of the data and the commit flag together in the same EEPROM block, as opposed to storing them in separate areas, so that writing the last of the data and the clearing the commit flag becomes one atomic operation.)

Optimisations and Improvements in the Algorithm

In addition to finding the bugs above, the systematic analysis of the code required to write the JML specifications also had the benefit of suggesting several optimisations and improvements to the code.

Efficiency Improvements

The method `DGeneration(Tag tag)` discovers the generation indices associated with a tag, and then returns the indices of the oldest and newest generation, as well as the number of generations. To better understand the implementation of this method, it was annotated with JML `assert` clauses. An `assert` clause can occur anywhere in a method body, and specifies a property that should hold at this point in the program. When doing runtime assertion checking, any violation of an `assert` clause will produce an error message.

Annotating the implementation of DGeneration(Tag tag) with assert clauses, we discovered that one for-loop could be removed, as the value it computed could already be computed directly from values already known.

Also, a redundant modulo operation % (i.e. one where the first argument will always be smaller than the modulus) was discovered in the implementation of DGeneration.

Interface Improvements

The four operations for writing to the Transacted Memory are:

- DWriteFirst
- DWriteUncommitted
- DWriteCommittedAddGen
- DWriteCommittedMaxGen

These operations have identical postconditions, and only differ in their preconditions. This raises the question whether it is not better to have a single method DWrite, which chooses the 'right' write operation and executes it. Indeed the original Z specification offers such a 'comprehensive' write operation, defined by way of a schema conjunction of the write operations listed above. However, this operation was forgotten in the development of the C cum SPIN code.

An unsatisfactory feature of the Transacted Memory as originally implemented in C is that if there is insufficient space to perform a write operation, it may be carried out only partially, resulting in an inconsistent state, without any warning. The informal specification of DWriteFirst in Table 2 does indeed say that its effect is undefined if insufficient space is available. The same can happen in the other write operations, although their informal specifications do not say this.

Our initial JML specifications for the write methods, e.g. the one in Figure 2, did not allow for this, and the runtime assertion checker warned about violations of them.

We improved the Java implementation so that a OutOfTransactedMemoryException is thrown in case insufficient space is available to perform a write operation. The JML specifications were adapted accordingly. For example, in the specification for DWriteUncommitted in Figure 2 we added

```
signals
 (OutOfTransactedMemoryException)
     DRead(tag).equals(is) &&
     ! ddata[tag.value].committed;
```

stating that the write operation won't happen at all in case an OutOfTransactedMemoryException is thrown.

Similarly, the operation DNewTag was adapted to throw an OutOfTagsException when no additional tag is available, rather than producing an undefined result in this case.

4.1 Future Work with these JML specs

We also translated the abstract Z specification given in [6] to Java/JML. This was not difficult, given that JML comes with a package org.jmlspecs.models that provides Java implementations of all the standard mathematical concepts used in the Z specification. For example, Figure 3 gives the JML translation of the Z specification of the operation *ACommit* shown in Section 2.

One obvious difference is that the Z specification looks prettier, as in Java/JML we do not have conventional mathematical notation, such as $\in$ or $\neq$.

A more important difference is that the JML/Java specification can be turned into an executable one, namely

```
public void ACommit(Tag t)
{ committed = committed.insert(t);
}
```

We could use this Java implementation of the abstract specification to give a more detailed

```
//@ import org.jmlspecs.models.*;

/*@ requires    assocs.domain().has(t) &&
  @             ! assocs.apply(t).isEmpty() ;
  @ ensures     committed.equals( \old(committed).insert(t));
  @*/
public void ACommit(Tag t)
```

Figure 3: JML specification of ACommit

specification for our current Java implementation. Basically, the idea would be to define a Java implementation which executes the current Java implementation and this more abstract one side by side, and express the relation between the two in JML assertions. However, as the abstract specification does *not* consider the possibility of card tears, the precise relation between this abstract implementation and the current Java implementation is not trivial to make precise. This is left as future work.

5 Conclusions

The work described in this paper, i.e.

- developing a Java implementation based on a C implementation, and
- developing JML specifications based on a Z specification, and
- checking the Java implementation against the JML specification using runtime assertion checking,

has been successful in finding bugs and improving the implementation. The bugs we found range from simple typos to more serious errors, and to some misunderstandings between different people that have been involved in the design of the Transacted Memory.

It is disappointing that the careful development of the system as reported in our previous paper [6] – starting from a formal abstract Z specification that was refined to an C/SPIN implementation, which was model-checked – did leave these bugs in the final implementation.

In all fairness, we must admit that the original testing scenario for the C/SPIN implementation with the model-checker SPIN was too restricted. Conventional testing of the C implementation would have discovered many of the bugs that we found, but probably with more effort. Runtime assertion checking of JML specifications makes it easier to locate bugs than conventional testing. Indeed, no complicated testing scenarios were needed to find any of the bugs discussed.

Some problems and possible improvements were found before we even tried runtime assertion checking, but were spotted when trying to come up with good specifications in the first place. Annotating Java code with JML specifications provides a systematic way of performing a thorough code review, which can help to discover bugs and may point to possible optimisations or improvements. By contrast, testing of the code may find the bugs, but will probably not suggest optimisations or improvements.

There is a fairly standard recipe for annotating Java code with JML. Typically, one starts by giving pre- and postconditions for each method; these can be based on existing informal specifications, on our informal understanding of the program, and – somewhat exceptionally here – on the formal Z specifications. For each method implementation one then informally checks that any method invocations it contains do not violate their preconditions; this may require further strengthening of its precondition, or the introduction of loop invariants. Then one compares the different pre- and postconditions that have been written. Commonalities between pre-

and postconditions may suggest class invariants. Differences between them may point out possible omissions; e.g if the precondition of `DWriteUncommitted (Tag tag)` requires a tag to be uncommitted, then its postcondition should probably state whether this tag remains uncommitted or not, and possibly other methods that have a tag as argument should be specified with similar conditions. Finally, any violations of assertions found during runtime assertion checking in test scenarios may of course lead to improvements in the JML specifications.

For the system we considered, a vital advantage of using Java over using C is that we can conveniently model card tears using Java's exception mechanism. A disadvantage of using Java instead of C is that C is probably closer to a realistic implementation in actual hardware.

Using Java and JML, rather than C and Z, for implementation and specification, has had several advantages.

Firstly, it becomes possible to check the relation between implementation and specification: runtime assertion checking tells us where Java implementation and JML specification disagree. This may of course just as well be a mistake in the Java implementation as a mistake in the JML specification.

Secondly, Java implementation and JML specification are close together, in the same file. The usefulness of this is illustrated by the fact that the Z specification actually discusses the possibility of bug 2, but in a footnote on page 34 of [2], something one is not likely to notice or remember when working on the C implementation.

Finally, the JML specifications are a lot easier to understand than the Z specifications, except for experts in Z. JML mainly uses Java notions and notations, and it has been the overriding design principle in the design of JML that specifications should be easy to understand by any Java programmer. Indeed, a point we would like to stress is that formal methods need not involve notations and tools that only specialists can use. Our formal model is a Java program, that can be understood by anyone familiar with Java, as can the formal specifications for it written in JML. In this respect, it is interesting to note the contrast with Z and SPIN – or indeed UML! Developing the kind of JML specifications we discussed in this paper and using the runtime assertion checker should not pose any problem for competent Java programmers.

6 Acknowledgments

The work by Erik Poll is financially supported by the IST Programme of the European Union, as part of "VerifiCard" project (IST-2000-26328).

References

[1] C.-B. Breunesse, B. Jacobs, and J. van den Berg. Specifying and verifying a decimal representation in Java for smart cards. In *9th Algebraic Methodology and Software Technology (AMAST)*, volume LNCS 2422, pages 304–318, St. Gilles les Bains, Reunion Island, France, Sep 2002. Springer-Verlag, Berlin.

[2] M. J. Butler, P. H. Hartel, E. K. de Jong, and M. Longley. Applying formal methods to the design of smart card software. Declarative Systems & Software Engineering Technical Reports DSSE-TR-97-8, Univ. of Southampton, 1997. `http://www.dsse.ecs.soton.ac.uk/techreports/97-8.html`.

[3] N. Cataño and M. Huisman. Formal specification of Gemplus's electronic purse case study. In L. H. Eriksson and P. A. Lindsay, editors, *Formal Methods: getting IT right – Formal Methods Europe (FME)*, volume LNCS 2391, pages 272 – 289, Copenhagen, Denmark, Jul 2002. Springer-Verlag, Berlin.

[4] Y. Cheon and G.T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). In Hamid R. Arabnia and Youngsong Mun, editors, *Proceedings of the International Conference on Software Engineering Research and Practice (SERP '02), Las Vegas, Nevada, USA, June 24-27, 2002*, pages 322–328. CSREA Press, June 2002.

[5] Eduard Karel de Jong and Jurjen Norbert Bos. *Arrangements Storing Different Versions of a Set of Data in Separate Memory Areas and Method for Updating a Set of data in a Memory.* Dutch Patent Application, PCT/NL99/00360, June 10, 1999. International Publication Number WO 00/77640, 2000. WIPO, Vienna.

[6] P. H. Hartel, M. J. Butler, E. K. de Jong Frz, and M. Longley. Transacted memory for smart cards. In J. N. Olivieira and P. Zave, editors, *10th Formal Methods for Increasing Software Productivity (FME)*, volume LNCS 2021, pages 478–499, Berlin, Germany, Mar 2001. Springer-Verlag, Berlin. `http://www.dsse.ecs.soton.ac.uk/techreports/2000-9.html`.

[7] P. H. Hartel and E. K. de Jong Frz. A programming and a modelling perspective on the evaluation of Java card implementations. In I. Attali and T. Jensen, editors, *1st Java on Smart Cards: Programming and Security (e-Smart)*, volume LNCS 2041, pages 52–72, Cannes, France, Sep 2000. Springer-Verlag, Berlin. `http://www.dsse.ecs.soton.ac.uk/techreports/2000-8.html`.

[8] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on software engineering*, 23(5):279–295, 1997. `http://cm.bell-labs.com/cm/cs/who/gerard/`.

[9] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Business and Systems*, pages 175–188. Kluwer Academic Publishers, Boston/Dordrecht/London, 1999.

[10] S. M. Nettles and J. M. Wing. Persistence+undoability=transactions. In *25th Hawaii System Sciences (HICS)*, volume 2, pages 832-843. IEEE Comput. Soc. Press., Los Alamitos, California, 1991.

[11] E. Poll, J. van den Berg, and B. Jacobs. Specification of the JavaCard API in JML. In J. Domingo-Ferrer and A. Watson, editors, *Fourth Smart Card Research and Advanced Application Conf. (CARDIS)*, pages 135–154, Bristol, UK, Sep 2000. Kluwer Academic Publishers, Boston/Dordrecht/London.

[12] E. Poll, J. van den Berg, and B. Jacobs. Formal specification of the JavaCard API in JML: the APDU class. *Computer Networks*, 36(4):407–421, Jul 2001. The JML specs of the JavaCard API 2.1.1 are available online at `http://www.cs.kun.nl/~erikpoll/publications/jc211_specs.html`.

[13] J. M. Spivey. *The Z notation.* Prentice Hall, Englewood Cliffs, New Jersey, 1989.

Model Checking of Multi-Applet JavaCard Applications*

Gennady Chugunov[1] Lars-Åke Fredlund[1] Dilian Gurov[2]

[1]Swedish Institute of Computer Science

[2]Department of Microelectronics and Information Technology, Royal Institute of Technology (KTH)

Abstract

The paper describes a framework for model checking JavaCard applets on the bytecode level. From a set of JavaCard applets we extract their method call graphs using a static analysis tool. The resulting structure is translated into a pushdown system for which the model checking problem for Linear Temporal Logic (LTL) is decidable, and for which there are efficient model checking tools available. The model checking approach of the paper is tailored to the analysis of inter applet (intra card) communications and we demonstrate it using a prototypical example of a purse applet and a set of loyalty applets.

1 Introduction

Smart cards have come to play an ever increasing role in our lives. We use them in electronic banking, to keep health care data, for mobile telephony, and in many other applications. The most important aspect of smartcards is their security; users and card issuers have to agree that the level of security provided by a smartcard platform is enough to prevent malicious agents from abusing their trust in a card application.

Since the number of smartcard applications is growing rapidly, it is natural to provide smartcards with the possibility of accommodating multiple applications, and the possibility to delete or add new applications after the card has been issued. Furthermore, such multi-application smartcards allow partner applications to cooperate and exchange data. Popular applications of multi-application cards are partner loyalty programs, mobile telephone to banking partnership programs, etc. The JavaCard platform [12] is one platform for building such multi-application smartcards. It is based on a subset of Java tailored to the task of embedding on a smartcard. The current standard omits many of the features of Java such as concurrency through threads, garbage collection, and many API functions but has a notion of applets to support multiple applications.

One important aspect which distinguishes multi-applet JavaCards from single-applet ones is the support for inter-applet communication via method calls. Communication naturally comes at a price: applets must guard against illicit invocations of their public methods from unwarranted applets, and from leakage of data to third parties. Even if a multi-applet application were to be proved safe, there still exists the possibility of new unsafe applets being loaded onto the card post-verification. The JavaCard platform provides features to partially address these security concerns. Apart from a Java-style byte code verifier, which in the current generation of JavaCard smartcards is typically located off-card, there is a concept of a communication firewall that by default prohibits applets from communicating with each other. To enable communication to flow between applets, a recipient applet has to explicitly permit calls from the caller applet.

Such checks as above are static in nature, e.g., method calls are always allowed, or they are never allowed. The work reported here in contrast permits to begin to characterise the temporal restrictions of inter-applet communications. In the formulation of such restrictions we consider a situation when a set of applets have been loaded onto a smartcard,

*The research has been conducted within the VerifiCard project with financial support from the IST programme of the European Union.

and formulate properties in Linear Temporal Logic (LTL) regarding inter–applet communications (in addition to properties about intra–applet method calls and API usage).

To provide a semantic bridge between multi-applet programs and the temporal logic specification language, we use the abstract notion of a program graph, capturing the control flow of programs with procedures/methods, and which can be efficiently computed. The behaviour of such program graphs is defined through the notion of pushdown systems, which provide a natural execution model for programs with methods (and possibly recursion), and for which completely automatic model checkers for LTL exist.

In more detail the model checking proceeds as follows. First the method call graphs of a set of JavaCard applets are obtained using a Java byte code analysis tool [13] developed at INRIA Rennes, which we have adapted for JavaCard. The analysis is performed on a class basis. As a consequence individual applet instances cannot be reasoned about; correctness properties concern activation of methods of classes extending the JavaCard `Applet` class, rather than activation of methods of an applet instance. Further details and limitations of this static analysis procedure are discussed in Section 2.

The resulting method call graphs are translated into pushdown systems, a natural execution model for programs with recursion. Essentially a pushdown system is a pair of a control location with a stack of stack symbols. In our encoding we use a single control location and let the stack symbols represent the program points of the underlying JavaCard applets. The details of the translation are elaborated in Section 3.1.

For pushdown systems the model checking procedure for Linear Temporal Logic (LTL) is decidable and of polynomial complexity in the size of the system [3, 9, 7]. The atomic predicates of the logic, tailored to JavaCard, are the program points themselves and predicates expressing class and package membership of program points. The Moped model checker [8] is used to check LTL properties of pushdown systems. Sections 3.2,3.3 and 3.4 describes the logic and our use of the Moped tool in further detail.

To motivate and demonstrate our approach we have selected a prototypical JavaCard example: a purse applet stores money, and interacts with loyalty applets on receiving a purchase order. A loyalty applet can have agreements with other applets, and can thus in turn communicate with another applet on receiving information about a purse transaction. In Section 4 we demonstrate the effectiveness of our approach in analysing such inter-applet communication patterns.

There exists by now a growing number of related work concerning model checking Java (or JavaCard), or more general formal analysis of JavaCard applications; below we will mention a few of them.

The Compaq Extended Static Checker for Java (ESC/Java) [14], developed at the Compaq Systems Research Center (SRC), is a programming tool for finding errors in Java programs. ESC/Java includes an annotation language with which programmers can express design decisions using light-weight specifications. Checking is neither sound nor complete, but can yield informative warning messages[1]. A case study in the context of JavaCard, based on the Gemplus purse applet, is presented in [5].

The first version of the Java PathFinder [10], JPF, was a translator from a subset of Java 1.0 to PROMELA, the programming language of the Spin model checker. A similar translator tool from Java to PROMELA (actually the variant of PROMELA for the dSpin tool) is reported in [11]. The Java Pathfinder tool is especially suited for analyzing multi-threaded Java applications, where normal testing usually falls short. The tool can find deadlocks and violations of boolean assertions stated by the programmer in a special assertion language. A second version of the tool reportedly works directly on bytecode and has support for garbage collection[2].

The Bandera Project [6] aims to develop techniques and tools for automated reasoning about Java based software system behavior, and to apply these tools to construct high-confidence mission-critical software. Automated reasoning is achieved by (1) mechanically creating high-level models of software systems using abstract interpretation and partial evaluation technologies, and then (2) employing model-checking techniques to automatically verify that software specifications are satisfied by the model[3].

[1]`http://research.compaq.com/SRC/esc/`
[2]`http://ase.arc.nasa.gov/jpf/`
[3]`http://www.cis.ksu.edu/santos/bandera/`

In [2] an approach is presented for checking properties of multi-applet interactions of JavaCards based on associating security levels to applets and applet data, and to thus detect illegal flow of information between applets. Technically the approach requires building abstract models by hand from byte code, and then to check them automatically using the SMV model checker.

Our work is related to the program verification approach of [13] which is based on method call graphs. The operational semantics of the graphs, however, is given there directly through a set of transition rules (rather than through pushdown systems), and security properties are expressed as call-stack invariants. Following a similar program representation, a compositional account is given in [1], where a compositional proof system for inferring temporal properties of a multi-applet program from the properties of the individual applets is presented.

2 Constructing Method Call Graphs

We use an external static analysis tool, developed for a Java verification framework [13], to generate call graphs which abstract from everything (such as data variables, and parameters to method calls) but the presence and order of method calls inside method bodies. The analysis tool performs a safe over-approximation (with regards to preservation of LTL safety properties) in the sense that call edges may be present in the result call graph even if they cannot be invoked at runtime, but the opposite does not hold. For instance, when the static analysis cannot determine which class method is invoked in a method call, typically due to subtyping, then a call edge is generated to a target method in every possible class, thus increasing the nondeterminism in the generated call graph. The static analysis tool generates graphs with information about exceptional behaviours. In this work exceptional edges, and nodes, are translated into nondeterministic constructs thus effectively increasing the non-determinism in program behaviour in a conservative fashion.

The call graph generation is also conservative with respect to the JavaCard firewall mechanism, which is not considered during static analysis. That is, a method call that at runtime will fail the security checks of the JavaCard runtime environment will nevertheless invariably be included in the method call graphs.

Analysis starts from a set of JavaCard classes, which should include the implementation of all on-card applets. To refine the analysis, and to permit analysis of JavaCard API usage, the API classes of SUN's Java Card Development Kit (version 2.1.2) are included in the method call generation. The result of analysis is a set of method call graphs.

2.0.1 Method Call Graphs

The methods M are partitioned into classes C, which are themselves partitioned into packages P. We assume the usual Java naming conventions with fully qualified names, i.e., a class has a name *Package.identifier* and a method has a name *Class.identifier*.

Definition 1 (Method Graph, adapted from [13]). A *method graph* is a tuple

$$m \triangleq (V_m, \to_m, \lambda_m, \mu_m)$$

such that:

(i) V_m are the *program points* of m,

(ii) $\to_m \subseteq V_m \times V_m$ are the *transfer edges* of m, and

(iii) $\lambda_m : V_m \to T$ designates to each program point of m a *program point type* from the set $T \triangleq \{\mathsf{entry}, \mathsf{seq}, \mathsf{call}, \mathsf{return}\}$.

(iv) $\mu_m : V_m \to \wp(M)$ designates to each program point of type call of m a non-empty set of methods.

We assume the program point sets V_m to be pairwise disjoint. The program points of the program is the set $V \triangleq \bigcup_{m \in M} V_m$.

The program point type indicates whether (`entry`) a node is the entry point of a method, (`seq`) a node in which no method call or return takes place, (`call`) a node from which a method call takes place, or (`return`) a node in which the execution of the method finishes and control flow returns to the calling method.

For convenience, we introduce the predicates

$$\begin{aligned}
v : t &\triangleq \lambda_m(v) = t \text{ for } t \in T \\
v : \mathsf{loc}\, m &\triangleq v \in V_m \\
v : \mathsf{entry}\, m &\triangleq v : \mathsf{entry} \wedge v : \mathsf{loc}\, m \\
v : \mathsf{return}\, m &\triangleq v : \mathsf{return} \wedge v : \mathsf{loc}\, m \\
v : \mathsf{class}\, c &\triangleq \exists m.\, v : \mathsf{loc}\, m \wedge m \in c \\
v : \mathsf{package}\, p &\triangleq \exists c.\, v : \mathsf{class}\, c \wedge c \in p
\end{aligned}$$

We further define a predicate $v : \mathsf{api}$, which holds if the program point v occurs in a method in a JavaCard API package (for standard JavaCard this corresponds to one of `java.lang`, `javacard.framework`, `javacard.security` or `javacardx.crypto`).

3 Model Checking Method Call Graphs

3.1 Pushdown Systems

Pushdown systems provide a natural execution model for programs with recursion. They form a well-studied class of infinite-state systems for which many important problems like equivalence checking and model checking are decidable [4].

Definition 2 (PDS, from [7]). A *pushdown system (PDS)* is a tuple

$$\mathcal{P} \triangleq (P, \Gamma, \Delta)$$

where:

(i) P is a finite set of *control locations*;

(ii) Γ is a finite set of *stack symbols*;

(iii) $\Delta \subseteq (P \times \Gamma) \times (P \times \Gamma^*)$ is a finite set of *rewrite rules* of the shape $\langle p, \gamma \rangle \dashrightarrow \langle q, \sigma \rangle$.

The set $P \times \Gamma^*$ are the *configurations* of $\mathcal{P}$. If $\langle p, \gamma \rangle \dashrightarrow \langle q, \sigma \rangle$ is a rewrite rule of $\mathcal{P}$, then for each $\omega \in \Gamma^*$ the configuration $\langle q, \sigma \cdot \omega \rangle$ is an *immediate successor* of the configuration $\langle p, \gamma \cdot \omega \rangle$. A *run* of $\mathcal{P}$ is a sequence $\rho = \langle p_0, \sigma_0 \rangle \langle p_1, \sigma_1 \rangle \langle p_2, \sigma_2 \rangle \cdots$, such that for all i, $\langle p_{i+1}, \sigma_{i+1} \rangle$ is an immediate successor of $\langle p_i, \sigma_i \rangle$.

We now define how a set of methods M induces a PDS.

Definition 3 (Induced PDS, formalising [8]). A set of methods M *induces* a PDS

$$\mathcal{P} \triangleq (P, \Gamma, \Delta)$$

as follows:

(i) P consists of the single control location p;

(ii) Γ is the set V of program points;

(iii) Δ is the set $\bigcup_{m \in M} \bigcup_{v \in V_m} \mathrm{Prod}(v)$, where $\mathrm{Prod}(v)$ is a set of rewrite rules defined as:

$$\begin{cases}
\{\langle p, v \rangle \dashrightarrow \langle p, v' \rangle \mid v \dashrightarrow_m v'\} \\
\quad \text{if } v : \mathsf{entry} \text{ or } v : \mathsf{seq} \\
\bigcup_{m' \in \mu_m(v)} \left\{ \begin{array}{l} \langle p, v \rangle \dashrightarrow \langle p, v' \cdot v'' \rangle \mid \\ \quad v' : \mathsf{entry}\, m', v \dashrightarrow_m v'' \end{array} \right\} \\
\quad \text{if } v : \mathsf{call} \\
\{\langle p, v \rangle \dashrightarrow \langle p, \epsilon \rangle\} \\
\quad \text{if } v : \mathsf{return}
\end{cases}$$

The rewrite rules of the pushdown system can be interpreted as simply manipulating the calling stack of the program from which the PDS was obtained. Given a configuration $c \equiv \langle p, v \cdot \sigma \rangle$ let point $(c) \triangleq v$.

3.2 Specification Language

Our specification language is linear temporal logic (LTL), with program point predicates p as atomic propositions but omitting the type predicate $v : t$. The choice of linear temporal logic as the specification language, instead of for instance the modal μ-calculus for which the model checking problem for our encoding into pushdown systems is also efficiently decidable, was solely motivated by the existence of the efficient model checker Moped [8] for LTL.

The operators of the logic are the standard ones. If ϕ and ψ are formulas then so are $\neg\phi$, $\phi \wedge \psi$, $\phi \vee \psi$, $\mathcal{X}\phi$ and $\phi\, \mathcal{U}\, \psi$. The meaning of formulas is defined with respect to runs of infinite length $r \equiv c_0 c_1 c_2 \ldots$. We let r_i denote the suffix of r starting in configuration c_i. Then satisfaction $r \models \phi$ of a formula ϕ by a run r is defined as:

$r \models p$	iff	$\mathsf{point}\,(c_0) : p$
$r \models \neg\phi$	iff	not $r \models \phi$
$r \models \phi \wedge \psi$	iff	$r \models \phi$ and $r \models \psi$
$r \models \phi \vee \psi$	iff	$r \models \phi$ or $r \models \psi$
$r \models \mathcal{X}\,\phi$	iff	$r_1 \models \phi$
$r \models \phi\ \mathcal{U}\ \psi$	iff	there is an $i \geq 0$ such that $r_i \models \psi$ and $r_j \models \phi$ for all $0 \leq j < i$

Henceforth let false abbreviate $p \wedge \neg p$ for some atomic predicate p, true abbreviate $\neg$false, $\phi \Rightarrow \psi$ abbreviate $\neg\phi \vee \psi$, and next ϕ abbreviate $\mathcal{X}\ \phi$ and ϕ until ψ abbreviate $\phi\ \mathcal{U}\ \psi$. Further define eventually $\phi \triangleq$ true $\mathcal{U}$ ϕ and always $\phi \triangleq \neg$(eventually $\neg\phi$). The weak until operator ϕ weakuntil ψ abbreviates ϕ until $\psi \vee$ always ϕ. Finally let never $\phi \triangleq$ always $\neg\phi$.

Given a PDS *pds* let the notation $m \vdash \phi$ express the judgement that all runs starting in the entry program point of the method m satisfy ϕ. More formally:

Definition 4 (Model Checking a Method Call). Given a PDS *pds* with the single control location p and a method m, the judgement $m \vdash \phi$ is valid iff for every run r of the PDS *pds'* from the initial configuration $\langle p, v \cdot m\_\mathtt{loop}\rangle$, $r \models \phi$ holds, where v is the entry program point of method m (i.e. v : entry m), and *pds'* is the PDS *pds* extended with the fresh stack symbol $m\_\mathtt{loop}$ and the single rewrite rule $\langle p, m\_\mathtt{loop}\rangle \rightarrow \langle p, m\_\mathtt{loop}\rangle$ to achieve infinite runs.

The definition of a judgement $m \vdash \phi$ is motivated by the Moped tool which implements an algorithm for checking an initial configuration against an LTL formula.

3.3 Specification Patterns

As in the Bandera project [6] specification patterns are used to facilitate formulating correctness properties. These specification patterns concern temporal properties of method invocations, and are either *temporal patterns* or *judgement patterns* concerning the invocation of a particular method. Below a set of patterns that we have defined, and which are commonly used, are given.

To express that *within the call of a method m the property ϕ holds* the judgment pattern

$$\text{Within } m\ \phi \triangleq m \vdash \phi$$

is used. The property that *a call to m_1 never triggers method m_2* can be specified as:

$$\begin{aligned} & m_1 \text{ never triggers } m_2 \\ & \triangleq \text{Within } m_1\ (\neg(\text{eventually loc}\, m_2)) \\ & \equiv \text{Within } m_1\ (\text{never loc}\, m_2) \end{aligned}$$

Next define the temporal patterns (formulas) (i) m_2 after m_1, i.e., m_2 can only be called after a call to m_1; (ii) m_2 through m_1, i.e., m_2 can only be called from m_1; (iii) m_2 from m_1, i.e., m_2 can only be called directly from m_1; and (iv) m_1 excludes m_1, i.e., when m_1 is called this excludes the possibility that m_2 will later be called; (v) p cannotCall m, i.e., the method m cannot be directly called from any method in package p.

$$\begin{aligned} & m_2 \text{ after } m_1 \\ & \quad \triangleq (\text{never loc}\, m_2) \text{ weakuntil loc}\, m_1 \end{aligned}$$

$$\begin{aligned} & m_1 \text{ excludes } m_2 \\ & \quad \triangleq (\text{eventually loc}\, m_1) \Rightarrow \text{never loc}\, m_2 \end{aligned}$$

$$\begin{aligned} & m_2 \text{ from } m_1 \\ & \quad \triangleq \text{always } (\neg(\text{loc}\, m_1 \vee \text{loc}\, m_2) \Rightarrow \text{next } \neg\text{loc}\, m_2) \\ & \qquad \wedge \neg\text{loc}\, m_2 \end{aligned}$$

$$\begin{aligned} & m_2 \text{ through } m_1 \\ & \quad \triangleq \neg\text{loc}\, m_2 \text{ weakuntil loc}\, m_1 \\ & \qquad \wedge \left(\begin{array}{l} \text{always return } m_1 \Rightarrow \\ \quad \text{next } (\neg\text{loc}\, m_2 \text{ weakuntil loc}\, m_1) \end{array} \right) \end{aligned}$$

$$\begin{aligned} & p \text{ cannotCall } m \\ & \quad \triangleq \text{always} (\text{package } p \Rightarrow \text{next } \neg\text{loc}\, m) \end{aligned}$$

The intuitive idea of the formulation of m_2 from m_1 is to express that the current program point can be in method m_2 only because of a direct call from m_1, or because it was already in m_2, and initially the program point is not in m_2.

The above patterns can be combined with the Within pattern. For example,

$$\text{Within } m_1\ (m_3 \text{ after } m_2)$$

expresses that during a call to m_1 the method m_3 will be called only after calling m_2.

An alternative technique for expressing correctness properties of behaviours of programs of stack-based languages is to use stack inspection techniques [13]. Essentially these techniques express constraints on the set of all possible runtime stacks. Note however that for instance the after property above cannot directly be coded as a stack inspection property since the calls to m_1 and m_2 need not be concurrent.

3.4 A Tool for Model Checking Pushdown Systems

The Moped tool [8] can check a pushdown system, from an initial configuration, against an LTL formula where the atomic predicates consists of a set of atomic symbols that checks the identity of the top stack symbol or the control location (i.e., simply checks name equality). In case the LTL formula is falsified a reduced pushdown system constructed from the original one, that also falsifies the LTL formula, is presented as diagnostic information.

To represent the non-identity atomic predicates (e.g., package, entry, ...) as "Moped LTL formulas" a number of options are possible. Consider for instance the package atomic predicate. A direct representation of the predicate in Moped LTL would consist of a disjunction over all the program points in any class in the package.

An alternative representation strategy is to enrich the translation from a call graph to a pushdown system. Since Moped provides boolean variables we could represent the current package identity encoded in a set of boolean variables in the pushdown system. These variables would then be updated for every rewrite rule that crosses package boundaries. Finally the representation of the package predicate itself would consist of a simple boolean condition.

We have instead opted to extend the Moped tool with atomic predicates that can match a control location, or the top stack symbol, against a regular expression. These predicates check the syntactic shape of the symbol being tested.

Consider the naming of program points of a method m by the call graph construction. Its entry program point will be named *m_entry*, its (unique) return program point will be named *m_exit*, and all other program points in m are of the form *m_n* where n is a natural number.

With these conventions in place the atomic predicates can be represented in "regular expression Moped" as indicated below:

$$\begin{array}{rcl}
\mathsf{loc}\ m & \triangleq & m\texttt{\_.*} \\
\mathsf{entry}\ m & \triangleq & m\texttt{\_entry} \\
\mathsf{return}\ m & \triangleq & m\texttt{\_exit} \\
\mathsf{class}\ c & \triangleq & c\texttt{\textbackslash..*\_.*} \\
\mathsf{package}\ p & \triangleq & p\texttt{\textbackslash..*\textbackslash..*\_.*}
\end{array}$$

In the encoding it is assumed that the dot symbol '`.`' has to be quoted using a backslash character inside a regular expression to represent itself, rather than representing any character.

So called wildcards can be used in a regular expression to achieve a limited form of quantification over program points. The static analysis tool, for instance, gives the name *p.c.*`<init>` to an object constructor method *p.c*. Thus, whether the current program point is in any object constructor can be tested by the regular expression predicate `.*\..*\.<init>_.*`. As a further example, the api predicate, which recognises control points inside an API function, can be defined
'`(java\.lang|javacard\..*|javacardx\..*).*`'.

4 Example

The model checking of JavaCard applets will be illustrated with an example; a modification of the purse example from SUN's JavaCard Development Kit (version 2.1.2). This example is a prototypical purse and loyalty smartcard application, which comprises around 1430 lines of JavaCard code.

To understand the example it is helpful to recall the execution characteristics of JavaCard applets (language version 2.1.1). An interaction with the card (after installation of an applet, and its selection) is initiated through calling its `process` method. Inter-applet communications, crossing package borders, are controlled by the JavaCard firewall mechanism and take place through special interface objects. The methods of such interface objects are indicated in Figure 1.

The purse applet keeps a balance that is updated upon requests from the environment. Purse transactions, whether successful or not, are logged to a transaction log. The operations of updating the balance, logging the new transaction and updating the

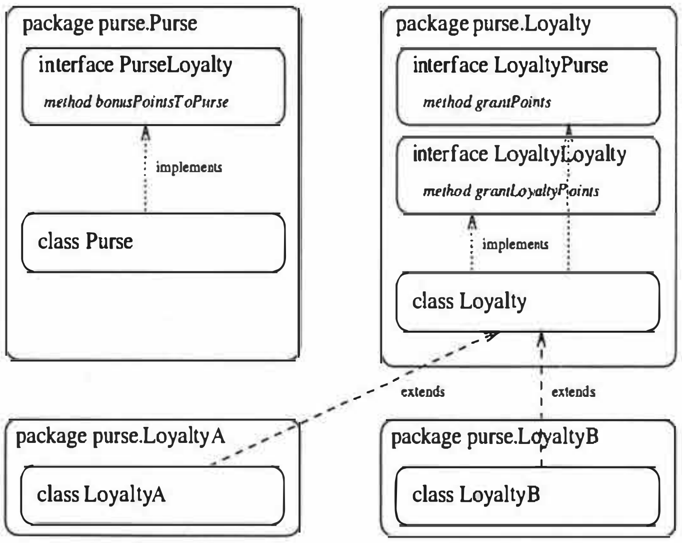


Figure 1: Purse Class Diagram

transaction number are made atomic through use of the transaction facility of JavaCard.

Upon completion of a new purse transaction the purse applet notifies subsidiary loyalty applets via the interface method `grantPoints`. These are applets that should be notified of the balance update so that they, for example, can award loyalty points. A concrete example is a bank smartcard with an embedded loyalty applet for a car rental company that awards bonus points for every car rented using the bank card.

In addition to these functionalities there are methods, accessible through the `process` method of the applet, for modifying most of the parameters of the purse applet, including adding knowledge about new loyalty applets that should be notified when card transactions occur.

The loyalty applets of the Development Kit purse application do not attempt to communicate with other applets. We have extended the example loyalty applet with two new functionalities: (i) A loyalty applet can have agreements with other loyalty applets to share bonus points; to achieve this we introduce direct loyalty applet to loyalty applet communication using the interface method `grantLoyaltyPoints`. (ii) loyalty applets can have an agreement with the purse to transfer, according to same fixed rate, part of the bonus points back to the purse. This is achieved through calling the interface method `bonusPointsToPurse` of the purse.

The modified purse and loyalties example is a rewarding example to study using our model checking approach as many key applet correctness properties can be phrased as properties of inter-applet communications.

4.1 Example Properties

Below we list a number of properties of the purse and loyalty applets, formulated using our judgement patterns. We introduce the following abbreviations of the applet class names:

$$\begin{aligned} purse &\triangleq purse.Purse.Purse \\ loyaltyA &\triangleq purse.LoyaltyA.LoyaltyA \\ loyaltyB &\triangleq purse.LoyaltyB.LoyaltyB \end{aligned}$$

Property 1: there are no calls to both grantPoints and grantLoyaltyPoints for the same applet. For all loyalty applets L it is the case that a call to *L.grantPoints* never triggers a call to *L.grantLoyaltyPoints*.

$$\begin{aligned} \phi_{1.1} &\triangleq loyaltyA.grantPoints \text{ never triggers } \\ & \quad loyaltyA.grantLoyaltyPoints \\ \phi_{1.2} &\triangleq loyaltyB.grantPoints \text{ never triggers } \\ & \quad loyaltyB.grantLoyaltyPoints \end{aligned}$$

Property 2: grantPoints is not transitive. For all loyalty applets L and L' it is the case that a call to L.*grantPoints* never triggers a call to L'.*grantPoints*. That is, the *grantPoints* method is neither transitive nor recursive.

$\phi_{2.1} \triangleq$ *loyaltyA.grantPoints* never triggers *loyaltyA.grantPoints*

$\phi_{2.2} \triangleq$ *loyaltyA.grantPoints* never triggers *loyaltyB.grantPoints*

$\phi_{2.3} \triangleq$ *loyaltyB.grantPoints* never triggers *loyaltyA.grantPoints*

$\phi_{2.4} \triangleq$ *loyaltyB.grantPoints* never triggers *loyaltyB.grantPoints*

Property 3: grantLoyaltyPoints is not transitive. The same as Property 2, but for *grantLoyaltyPoints*.

Property 4: grantLoyaltyPoints is called only through grantPoints. That is, within all purse methods m accessible from outside the card, the method L.*grantLoyaltyPoints* of a loyalty applet L is called only through a call to L'.*grantPoints* of another loyalty applet L' and never directly by the purse applet.

$\phi_{4.1} \triangleq$
Within m
loyaltyA.grantLoyaltyPoints through *loyaltyB.grantPoints*

$\phi_{4.2} \triangleq$
Within m
loyaltyB.grantLoyaltyPoints through *loyaltyA.grantPoints*

Property 5: Bonus point are awarded at most once within a transaction. Transfer of bonus points from a loyalty to the purse does not cause further bonus points to be awarded.

$\phi_5 \triangleq$
Within *purse.bonusPointsToPurse*
package *purse.Purse* $\vee$ api

That is, calls to the *bonusPointsToPurse* method does not cause a context switch to any other applet package (but possibly to the JavaCard Runtime Environment – JCRE).

The previous correctness properties were specific to certain applications whereas the following express properties that can be beneficial for any JavaCard applet.

Property 6: no constructors called. For all applets A it is the case that no constructor method is called within a call to A.*process*. This can be a crucial property for applets due to the absence of garbage collection in standard JavaCards. Let constructor express the regular expression predicate `.*\..*\.<init>_.*` which tests whether the current location is in a constructor method.

$\phi_{6.1} \triangleq$ Within *purse.process* ¬constructor

$\phi_{6.2} \triangleq$ Within *loyaltyA.process* ¬constructor

$\phi_{6.3} \triangleq$ Within *loyaltyB.process* ¬constructor

This property holds of the loyalty applets, but not of the purse applet which can create a new object during a call to the process method (without bad consequences, due to conditions involving data).

Property 7: recursion freeness. For all non-API methods m it is the case that a call to m never triggers another call to m.

$$\phi_7 \triangleq m \text{ never triggers } m$$

The elapsed time to construct the set of call graphs from the example classes was approximately 16 seconds on a Linux workstation with a Pentium III 1.9 GHz CPU and 256 MB of memory. The resulting call graphs, which includes API program points, consists of 2034 nodes and 3747 edges. The pushdown system generated from these call graphs has approximately 1200 production rules. To check the pushdown system against each of the formulas above, given an initial configuration, took less than one second on the same computer hardware as used for call graph generation.

5 Conclusions and Future Work

The paper proposes a framework for automatic model checking of temporal constraints on inter-applet communications in multi-applet JavaCards.

The framework has been realised by combining a class–based static analysis tool with an automatic model checker for pushdown system and linear temporal logic.

In the future we will refine the static analysis to permit the analysis of communication capabilities of single applets thus connecting to the work on compositional proof systems for JavaCard applets suggested in [1]. This will permit us to analyse whether an applet can operate safely on a smart card even when the knowledge about other applets on the card is imperfect.

Further information regarding the model checking framework and the availability of the tool components and examples can be obtained at the web location `http://www.sics.se/fdt/projects/VeriCode/`.

6 Acknowledgment

We would like to thank Mads Dam for his insightful comments on the verification of JavaCard applets, and Florimond Ployette of INRIA Rennes for his help with modifying the Java based static analysis tool for JavaCard. Thanks are due also to Stefan Schwoon from Technische Universität München for his assistance with the use of the Moped tool, and to the anonymous referees for their valuable remarks on the submitted version of this paper.

References

[1] G. Barthe, D. Gurov, and M. Huisman. Compositional verification of secure applet interactions. In *Proc. FASE'02*, volume 2306 of *Lecture Notes in Computer Science*, pages 15–32. Springer, 2002.

[2] P. Bieber, J. Cazin, P. Girard, J.-L. Lanet, V. Wiels, and G. Zanon. Checking secure interactions of smart card applets. In *ESORICS*, pages 1–16, 2000.

[3] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *Proc. CONCUR'97*, volume 1243 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 1997.

[4] O. Burkart, D. Caucal, F. Moller, and B. Steffen. Verification on infinite structures. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, pages 545–623. North Holland, 2000.

[5] N. Cataño and M. Huisman. Formal specification of Gemplus' electronic purse case study. In *Proc. FME'02*, Lecture Notes in Computer Science. Springer, 2002. To appear.

[6] J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, and H. Zheng. Bandera: extracting finite-state models from Java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.

[7] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *Proc. CAV'00*, volume 1855 of *Lecture Notes in Computer Science*, pages 232–247. Springer, 2000.

[8] J. Esparza and S. Schwoon. A BDD-based model checker for recursive programs. In *Proc. CAV'01*, volume 2102 of *Lecture Notes in Computer Science*, pages 324–336. Springer, 2001.

[9] A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. In *Electronic Notes in Theoretical Computer Science*, volume 9, 1997.

[10] K. Havelund and T. Pressburger. Model checking Java programs using Java pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[11] R. Iosif, C. Demartini, and R. Sisto. Modeling and validation of Java multithreading applications using spin, 1998.

[12] JavaCard 2.1.1 Documentation. Technical report, Sun Microsystems, May 2000. `http://java.sun.com/products/javacard/-specs.html#211`.

[13] T. Jensen, D. Le Metayer, and T. Thorn. Verification of control flow based security properties. In *IEEE Symposium on Security and Privacy*, pages 89–103, 1999.

[14] R. Leino, G. Nelson, and J. Saxe. ESC/Java User's Manual. Technical Report 2000–004, Compaq Systems Research Center, October 2002.

Secure Object Flow Analysis for Java Card

Marc Éluard and Thomas Jensen
{eluard,jensen}@irisa.fr

IRISA, Campus de Beaulieu
F-35042 Rennes, France

Abstract

The access control exercised by the Java Card firewall can be bypassed by the use of shareable objects. To help detecting unwanted access to objects, we propose a static analysis that calculates a safe approximation of the possible flow of objects between Java Card applets. The analysis deals with a subset of the Java Card bytecode focusing on aspects of the Java Card firewall, method invocation, field access, variable access, shareable objects and contexts. The technical vehicle for achieving this task is a new kind of constraints: quantified conditional constraints, that permits us to model precisely the effects of the Java Card firewall by only producing a constraint if the corresponding operation is authorized by the firewall.

1 Introduction

The Java Card language is a subset of Java, tailored to the limited resources available on today's smart cards. Java Card keeps the essence of Java, like inheritance, virtual methods, overloading, but leaves out features such as large primitive data types (`long`, `double` and `float`), characters and strings, multidimensional arrays, garbage collection, object cloning, security managers [1, 10]. Given the security-critical application areas of Java Card, the language has been endowed with an elaborate security architecture. *A priori*, applets are separated by a *firewall* which prevents one applet from accessing objects owned by another applet. Thus, even if a foreign applet obtains a reference to an object with confidential information this does not imply that the information is leaked. In order to provide a means of communication between separated applets, objects can be marked as *shareable*. This allows to grant access to (a subset of) the methods of the objects through the firewall. The problem is that marking an object as shareable means that its shared methods can be accessed by *all* applets that manage to get a reference to the object. To counter this problem, Java Card offers a limited form of stack inspection, allowing a "server" applet to know the identity of a "client" object which invoked a particular method. This, however, must be programmed explicitly by the application programmer. These mechanisms (described in detail in section 2) allow the design of secure applications but do not themselves guarantee security. Further code analysis must be employed to establish that the checks programmed in the server applet guarantee that confidential data is not leaked via shared objects. To summarize:

> The Java Card firewall can be bypassed by using shareable objects. Data flow analysis permits to calculate a safe approximation to the access control actually implemented by a set of applets, and thus to verify that a given access policy is respected.

This paper presents a flow analysis for Java Card programs. The analysis is *constraint-based* in that for each instruction of the program it generates a set of constraints describing the data flow of the instruction. The resolution of this system permits to find the possible values of the variables used in the program and the called method. The analysis relies on a novel technical device, *quantified conditional constraints* (*QCC*s), that allows to generate the set of constraints of a program *on demand*. This way of generating constraints is useful and natural when analyzing object-oriented languages where the control flow and the data flow are inter-dependent. It generalizes the conditional constraints proposed by Palsberg and Schwartzbach [20] for object-oriented type analysis.

The paper is organized as follows. Section 2 introduces the central features of the Java Card 2.1.1 firewall and provides a detailed example. Section 3 defines our representation of the Java Card bytecode. The abstract domains used in the analysis are given in Section 4 and Section 5 defines the set of quantified conditional constraints generated for each type of instruction. Section 6 shows how these *QCC*s can be solved iteratively and Section 7 shows how the analysis performs on the example from Section 2. Section 8 and Section 9 discuss related works and directions for extending this work.

2 The Java Card firewall

The Java Card platform is a multi-application environment in which an applet's sensitive data must be protected against malicious access. In Java, this protection is achieved using class loaders and security managers to create private name spaces for applets. In Java Card, class loaders and security managers have been replaced with the Java Card firewall. The separation enforced by the firewall is based on the Java Card's package structure (the same as Java's) and the notion of *contexts* (in Java Card, this notion is called *group context*).

When an applet is created, the *Java Card Runtime Environment* (`JCRE`) assigns it a unique applet identifier (AID). If two applets are instances of classes coming from the same Java Card package, they are said to belong to the same context, identified by the package name. In addition to the contexts defined by the applets executed on the card, there is a special "system" context, called the `JCRE` context. Applets belonging to this context can access objects from any other context on the card. Thus, the set of Java Card contexts is defined by:

Java Card contexts =
$\{ \texttt{JCRE} \} \uplus \{ \textit{pckg: a package name} \}$

Every object is assigned a unique *owner context viz.*, the context of the applet which created the object. A method of an object is said to execute in the context of its owner[1]. It is with this context that the `JCRE` determines whether an access to another object will succeed. The firewall isolates the contexts in the sense that a method executing in one context cannot access any fields or methods of objects belonging to another context.

There are two ways for the firewall to be bypassed: via `JCRE` entry points and via shareable objects. `JCRE` entry points are objects owned by the `JCRE` that have been specifically designated as objects accessible from any context. The most prominent example is the *Application Protocol Data Unit* (APDU) buffer in which commands sent to the card are stored. This object is managed by the `JCRE`, and in order to allow applets to access this object, it is designated as an entry point. Other entry points can be the elements of the table containing the AIDs of the applets installed on the card. Entry points can be marked as *temporary*. References to temporary entry points cannot be stored in objects (this is enforced by the firewall).

Two applets in different contexts may want to share some information. Java Card offers a sharing mechanism, called *shareable objects*, that gives limited access to objects across contexts. An applet can allow another applet to access an object's methods from outside its context. The mechanism is restricted to methods and cannot be applied to fields. It uses a shareable interface, that is an interface which extends `javacard.framework.Shareable`. In this interface, the applet gives the list of the method's signatures it wants to share. The class of the object to share must implement this interface. The "server" applet defines a method, `getShareableInterfaceObject`, called when an applet is asked to provide a shared object. The method receives the AID of the "client" applet which requested the shared object. Based on this information, the server decides what to return to the client, thus it is possible to share different objects with different client applets.

2.1 An example using shareable objects

Figure 1 contains an example illustrating the sharing mechanisms of the firewall. We have 3 applets: Alice, Bob and Charlie. Alice imple-

[1] In the case of a static call, the execution is in the caller's context.

ments a shareable interface MSI (we assume an interface MSI that extend Shareable in which the signature of the method foo is given) and is prepared to share an object MSIO (an instance of the class that implements the interface MSI) with Bob. When Alice receives a request for sharing (via a call to her method getSIO[2]) by the JCRE, she verifies that the caller is Bob. If it is Bob, she returns MSIO else she returns *Null*.

Bob can ask for a shareable object from Alice using the JCRE method getASIO[3]. Assume now that Bob (inadvertently) leaks a reference to MSIO to the third applet Charlie. Since the firewall only checks that the object is shared before granting access, Charlie can invoke the same methods of the MSIO object as Bob. Alice knows this so she decides to verify, at each access to one of her shared methods, the identity of the caller. Java Card offers a method for obtaining the AID of the context in operation before the last context switch, here called getPrevCxt[4]. Using this information Alice can discover when applets from contexts other than Bob's attempt to access the MSIO object.

2.2 Limitations of the firewall

The Java Card firewall has several shortcomings, as analysed in detail by Montgomery and Krishna [18]. One potential difficulty with the Java Card firewall is that shareable objects can be accessed by any applet and not only by the applet to which the reference was given, as illustrated by the example above. Since references can be passed from one applet to another, this opens up the possibility for methods in shared objects to be invoked by applets other than those for which they were intended. To protect applets against unwanted access, Java Card offers a limited form of the stack inspection mechanism that underlies the Java 2 security architecture. The system method getPrevCxt can be called to get access to the last context switch that took place. When a method is called from another applet, this context switch indicates the identity of the caller. This information can then be used to decide what value the method should return to the caller. It is, however, up to the programmer to implement this correctly. If the security mechanisms provided by the language are not used properly, unwanted information flow can arise as a result of objects flowing from one applet to another. In order to verify the access control actually implemented by a set of Java Card applets we have developed a static analysis that calculates, for each variable in a program, an approximation of the set of values that will be stored in this variable. This static approximation allows

- to signal potential data flow between applets that violates a given access control policy,
- or, if no such flow is detected, to provide a proof that all data flow respects the policy.

The analysis is based on a constraint-based type analysis for Java-like languages, but is modified to keep an accurate account of the Java Card specificities (like context and firewall). Indeed, since the security of an applet to a large extent relies on the use of the getPrevCxt method, the analysis must be able to model calls to this method precisely.

3 A representation of Java Card bytecode

To simplify the presentation, we work with a "three-address" representation of Java Card bytecode where arguments and results of an instruction are fetched and stored in local variables instead of being popped and pushed from a stack. This format is similar to the intermediate language *Jimple* used in the Java tool *Soot* [23] and the transformation of code into this format is straightforward. We furthermore assume that the constant pool has been expanded *i.e.* that indices into the constant pool have been replaced by the corresponding constant. For example, the bytecode instruction invokevirtual takes as parameter the signature of the method called, rather than an index into the constant pool. The formal representation of Java Card bytecode can be found in [17].

[2]In reality, this method is called getShareableInterfaceObject and is invoked by the JCRE that mediates all requests for shared objects.

[3]In reality, the method JCSystem.getAppletShareableInterfaceObject.

[4]In reality, this method is called JCSystem.getPreviousContextAID.

```
public interface MSI extends Shareable {
   Secret foo (); }
```

```
public class Alice extends Applet implements MSI {
  private Secret ObjectSecret;
  public Shareable getSIO (AID Client) {
    if (Client.equals (BobAID))
      return (this);
    return null; }
  public Secret foo () {
    AID Client;
    Secret Response;
    Client = getPrevCxt();
    if (Client.equals (BobAID))
      Response = ObjectSecret;
    return Response; } }
```

```
public class Bob extends Applet {
  public static MSI AliceObj;
  public void bar () {
    AliceObj = (MSI) getASIO
      (AliceAID); } }
```

```
public class Charlie extends Applet {
  private static MSI AliceObj;
  private static Secret AliceSecret;
  public void bar () {
    AliceObj = Bob.AliceObj;
    AliceSecret = AliceObj.foo (); } }
```

Figure 1: Example of shareable objects

3.1 Notations

The term $\mathcal{P}(X)$ denotes the power set of X: $\mathcal{P}(X) \equiv \{S \mid S \subseteq X\}$. A product type $X = A \times B \times C$ is sometimes treated as a labeled record: with an element x of type X, we can access its fields with the names of its constituent types ($x.A$, $x.B$ or $x.C$). A list is defined by enumeration of its elements: $x_1 :: \cdots :: x_n$. List elements can be directly accessed giving their position ($v(i)$ for the i^{th} element). Lists can be concatened: $(x_1 :: \cdots :: x_n) ::: (x_m :: \cdots :: x_p) = x_1 :: \cdots :: x_n :: x_m :: \cdots :: x_p$. X^* denotes the type of finite lists, whose elements are of type X. The symbol $\rightharpoonup$ is used to form the type of partial functions: $X \rightharpoonup Y$. The $\bar{v} \in \bar{E}$ notation denotes the formula $v_1 \in E_1 \wedge \cdots \wedge v_n \in E_n$.

3.2 Abstract syntax

Our program representation is a modified version of that of Bertelsen [5, 6]. We use Id_p, Id_{ci}, Id_f and Id_m to denote the set of qualified name of a package, of a class or an interface, of a field and of a method, respectively[5]. Id_v is the set of (unqualified) names of variables. To extract name information from an identifier, we use the notation $\lceil Id \rceil^x$, where Id is a qualified name and x the type of the projection[6]. We assume a set AID which contains the possible applet identifiers of the applets installed on a card. This set contains a special AID, written JCRE, for the Java Card Runtime Environment.

[5]The qualified name of an entity is the complete name. For a class, it is $p.c$ where p is the name of the package and c the (unqualified) name of the class. For a method ($c.m$) or a field ($c.f$)it is the qualified name of the class and the (unqualified) name of the method or field.

[6]To extract a (unqualified name), we use p for a package, c for a class or an interface, m for a method and f for a field. To extract a qualified name, we combine the symbols so, for example, $\lceil Id \rceil^{p.c}$ will extract a qualified name of a class (or interface) from the qualified name Id.

Classes and Interfaces A class or an interface descriptor consists of a set of the access modifiers ($\mathcal{P}(Mod_{ci})$), the name of the class or interface (Id_{ci}), the name of the direct superclass or the names of direct super-interfaces (*Ext*), the name of the interfaces that the class implements (*Imp*), the name of its package (Id_p), field declarations (*Fld*), method declarations and implementations (*Mtd*). A class must have one superclass, the default being `java.lang.Object`, but an interface can have zero or more super-interfaces. Only a class can implement an interface, so for an interface this set is empty. The fields are described by a map from field names (Id_f) to a pair consisting of a set of access modifiers ($\mathcal{P}(Mod_f)$) and a type descriptor (*Type*). The type of a field is either a primitive type (`boolean`, `short`, `byte`, `int`) or the name of a class or an interface. All of this information are stored in the class hierarchy (E_{ci}).

Methods The methods are described by a map that to a method signature (*Sig*) associates a method descriptor (Desc_m). This structure consists of a set of access modifiers ($\mathcal{P}(Mod_m)$), the code of the method (*Code*), a description of the formal parameters (*Param*), optionally a description of the variable used to return a value (*Res*) and the local variables of the method (*Varl*). A signature is the name of the method (Id_m) and the list of type descriptors for its parameters ($Type^*$). Code is a list whose elements consist of a program counter value (*Pc*[7]) and the instruction at this address (*Bytecode*). The set of local variables is the list of all variable names (Id_v) with their type descriptor (*Type*).

Bytecode Due to space limitations, in this paper, we only consider a subset of Java Card byte code. The subset is nevertheless sufficient to illustrate the different features of our analysis; see [16] for a treatment of the full language. In the following, T_i range over local variables and S_i is used to give the list of the type of the parameters for a call (which can be found in the constant pool).

The main departure from standard bytecode is the introduction of the construct `ifAID` $T \in S$ *BCinst*. This specialized if-instruction takes as argument a variable T that contains an AID, a set $S \in \mathcal{P}(\texttt{AID})$ and executes the instruction `BCinst` if the AID belongs to the set S. We have introduced this instruction to make explicit how the analysis takes information about AIDs into account. Ordinary bytecode can be transformed to use the `ifAID` instruction by identifying those conditional instructions that make test of the form $Aid \in S$. Most of such tests are syntactically explicit in Java Card source programs or can be identified by simple intra-procedural flow analysis.

$$Bytecode = \texttt{ifAID}\ T \in S\ BCinst \mid BCinst$$

The Java Card bytecode is transformed into a "three-address" like language. We will not describe this program transformation any further.

$BCinst =$
$$\begin{array}{ll} & T := \texttt{getstatic} f \\ \mid & T_0 := \texttt{invokeinterface}\ m\ T_1 \\ & T_2 \cdots T_n\ S_2{::}\cdots{::}S_n{::}S_{n+1} \\ \mid & T := \texttt{invokestatic}\ getPrevCtx \\ \mid & T_1 := \texttt{load}\ T_2 \\ \mid & T := \text{new}\ Id_c \\ \mid & \texttt{putstatic} f\, T \\ \mid & T_1 := \texttt{store}\ T_2 \end{array}$$

T:=`getstatic`f loads the value contained in the static field f of the class $[f]^{p.c}$ and stores it in T. T_0:=`invokevinterface` m T_1 $T_2 \cdots T_n$ $S_2{::}\cdots{::}S_n{::}S_{n+1}$ invokes the interface method m with the signature $S_2{::}\cdots{::}S_{n+1}$ on the object contained in T_1 with parameters $T_2 \cdots T_n$ and the result is stored in the variable T_0 with type S_{n+1}. T:=`invokestatic` *getPrevCtx* retrieves the AID of the last active context before the last context switch and stores it in T. T_1 := `load` T_2 loads the value contained in T_2 and stores it in T_1. T:=new C stores a reference to the object created at this program point in T. `putstatic`$f\,T$ loads the value contained in the variable T and stores it in the static field f of the class $[f]^{p.c}$. T_1 := `store` T_2 loads the value contained in T_2 and stores it in T_1.

[7]We assume furthermore a set *Pc* of program counters. A program counter identifies an instruction within the whole class hierarchy and not just a method.

3.3 Auxiliary functions on the class hierarchy

We define three predicates to determine if a class member (the second parameter) is visible from a given instruction (the first parameter). We have *CI_Visibility?* for a class or an interface, *Method_Visibility?* for a method and *Field_Visibility?* for a field. We must keep this test in the constraint because in some cases, like for the modifier `protect`, we need information about its dynamic values.

CI_Visibility?:
$$Id_c \times Id_{ci} \times E_{ci} \dashrightarrow Boolean$$
Method_Visibility?:
$$Id_c \times Id_c \times Desc_m \times E_{ci} \dashrightarrow Boolean$$
Field_Visibility?:
$$Id_c \times Id_f \times E_{ci} \dashrightarrow Boolean$$

The function *Lookup* models the dynamic search of methods underlying the virtual method calls. It takes as arguments the signature of a method, the class in which the method is declared, the class in which the invocation are made and the class hierarchy. It returns a set of fully qualified method names of the implementations of the method designated by the signature.

$$Lookup\colon Sig \times Id_{ci} \times Id_{ci} \times E_{ci} \dashrightarrow \mathcal{P}(Id_m)$$

A full description of the Java visibility rules and method resolution would be quite lengthy due to the non-trivial semantics of these two language features. We refer instead to the literature [12, 15, 14].

4 Abstract domains

Owners and contexts An object is owned by an applet (or the JCRE) thus an owner is uniquely identified by an AID. Since an AID does not directly specify the package to which the applet belongs, we add this information for convenience. Thus, the set of object owners is defined by:

$$Owner = Id_p \times \mathtt{AID}$$

We define an abstract context to be an abstraction of the call stack in which a method is executed (these contexts should not be confused with the Java Card notion of context). Our abstract contexts are designed to provide exactly the information that can be obtained by a call to the stack-inspecting method getPrevCxt (*cf.* Section 2). More precisely, the abstract context in which a method m is analyzed consists of a pair *(Prev,App)* where the first component *Prev* is the last active Java Card context before the last context switch and the second component *App* is the Java Card context of the *caller* (*i.e.*, the active context that invoked m). Formally we define:

$$Context = Owner \times Owner$$

Values We are primarily interested in modeling the object structure and ownership so we abstract primitive values such as booleans and integers to their type. To model the heap of objects, we adopt a common approach (going back to at least [13]) in which all objects created by the same new instruction are identified by one object. We refine this by keeping the owner as part of the abstract object. More precisely, a reference (*Ref*) to an object (*Obj*) is abstracted into the instruction that created the object and the owner of the object. We suppose we have a special *Null* reference.

$$Ref = (Pc \times Owner) \uplus \{ Null \}$$

We have three kinds of abstract values: references, applet identifiers and primitive values which as mentioned above are abstracted by their type.

$Value = Ref \uplus \mathtt{AID} \uplus$
$\quad \{\mathtt{boolean, short, byte, int}\}$

Concerning the concrete value in memory, we can have a class instance (*Obj*) which contains the name of the class (Id_{ci}), the owner of this instance (*Owner*), boolean flags indicating whether or not it is a JCRE entry point or a temporary JCRE entry point (*cf.* Section 2) and the set of fields (*Fldv*), a function which maps a field name to a set of values.

$Obj =$
$\quad Id_{ci} \times Owner \times JCREep \times tJCREep \times Fldv$
$Fldv =$
$\quad Id_f \rightarrow \mathcal{P}(Value)$

Firewall checks The checks made by the firewall are formalized through a collection of predicates. Covering all bytecode instructions would require eight different predicates ([16]); in this paper, we only use two of these predicates:

- The predicate *AccessInterface?* validates the access to methods of an object.

  AccessInterface?:
  $\quad Ref \times Ref \times Id_i \times E_{ci} \rightarrow boolean$

  The first reference represents the current context, the second represents the object on which the call is made and Id_i is the name of the interface which declared the method called. The access is authorized if and only if the context represented by the first reference is the context of the JCRE or if the contexts of the two references are the same or if the second reference represents a JCRE entry point or if the class of the object represented by the second reference implements a shareable interface and Id_i extends a shareable interface.

- The predicate *AccessPutstatic?* checks the validity of the access to a static field of a class.

  AccessPutstatic?:
  $\quad Ref \times Value \rightarrow boolean$

  The reference represents the current context that wants to store the value in the static field. The access is only authorized if the Java Card context represented by the reference is the context of the JCRE or if the value is neither a global object nor a temporary JCRE entry point.

5 Flow analysis

In this section we describe a data flow analysis to approximate the part of a program's behaviour relevant to security verifications. The main information calculated by our analysis is an approximation of the objects stored in the variables of the program. More precisely, we calculate the following information:

- $\mathcal{V}$ $[\![var;m,ctx]\!] \in \mathcal{P}(Value)$: the set of values stored in the variable *var* of method *m* when this method is called in context *ctx*.
- $\mathcal{SF}$ $[\![Id_{ci}]\!] : Id_f \dashrightarrow \mathcal{P}(Value)$: the possible values of the static fields of a given class.
- *mem* : $Ref \rightarrow Obj$: an approximation of the memory in which an abstract reference of form $(pc, owner)$ is mapped to an abstract object that safely approximates all those concrete objects allocated by instruction at address pc and owned by $owner$.
- $\mathcal{C}$ $[\![m,ctx]\!] \in \mathcal{P}(Ref)$: the set of objects on which a call to method *m* in context *ctx* is made.

It is important to analyze methods for each calling context since this is the information available to the firewall at run-time. An analysis that does not exactly model this information would have poor precision. This information serves two purposes: it permits constructing a control flow graph (by resolving which method is called at a given virtual method call) and it makes explicit if an object owned by an applet is stored in a variable accessible by another applet.

An intra-procedural analysis is required in order to approximate the behaviour of each server applet when it receives a request for a shared object. This analysis is orthogonal to the analysis presented in this paper and will not be described here. We shall assume the function:

$$Return\_SIO: \texttt{AID} \times \texttt{AID} \rightarrow \mathcal{P}(Ref)$$

It takes the AID of a server and the AID of a client and returns a safe approximation of the set of objects that the server accept to share with the client (the set that it returns is equal to or bigger than the set returned during the execution).

5.1 Quantified conditional constraints

The analysis will be specified in constraint-based style. We introduce a new type of constraints, the *quantified conditional constraints* (*QCCs*) that can be considered as a constraint scheme from which actual constraints can be generated.

The first kind of constraints used in static analysis is the simple constraint (*SC*). It is used to model the flow and the modification of information. A simple constraint has the form:

$$\text{Expression} \subseteq \text{Variable}$$

An extension of this kind of constraint was used by Palsberg and Schwartzbach [20] for type analysis. They take a simple constraint and add a condition under which the constraint is valid. Such a *conditional* constraint has the form:

$$\text{Class} \in \text{Variable}_1 \dashrightarrow \text{Expression} \subseteq \text{Variable}_2$$

The Variable_2 have Expression as possible value if and only if Class is a possible value for Variable_1. The simple constraint models an instruction of a method and the condition model the fact that this method can effectively be called.

This kind of constraints solves the problem that the constraints to be generated depend on the actual data flow of the program. The solution has the drawback that it has to generate all possible constraints from the outset and then test for each iteration and for each constraint whether it should be taken into consideration. In the following, we propose to generate the constraints set in an incremental fashion where constraints are only added once the data flow analysis has actually established that the constraints will be activated.

We propose to extend this kind of constraints in the following two ways:

- allow more conditions, to model, for example, the activities of the environment like the firewall checks or the visibility rules,
- produce dynamically the system based on the current value of each variable (instead of generating constraints for all possible values of the domain of the variable).

This new kind of constraints is called *quantified conditional constraints* and has the form:

$$\forall v_1, \cdots, v_n \in S_1, \cdots, S_n : \\ cond(v_1, \cdots, v_n) \rightarrow \\ cstr(v_1, \cdots, v_n)$$

Here, *cstr* is a set of simple constraints parameterized on $v_1, \cdots, v_n$ and *cond* are conditions on the values $v_1, \cdots, v_n$. Evaluation of such a *QCC* results in a set of constraints for each value $v_1, \cdots, v_n \in S_1, \cdots, S_n$ satisfying the condition *cond*. In our analysis, the *QCC*s have a particular structure, as shown below.

- The set S, used in the quantification, can be the set of possible values of a variable ($\mathcal{V}\ [\![x,m,ctx]\!]$), the set of objects on which a call is made ($\mathcal{C}\ [\![m,ctx]\!]$), the result of the *Lookup* or a constant set.
- The condition *cond* is a conjunction of conditions. It can be a test on the visibility, a firewall check or a test for membership of a constant set.
- A constraint *const* is a set of simple constraint *SC*. *SC* have a form: $Exp \subseteq Var$. *Exp* can be a variable, a constant set, a dereferencing of the memory, the set of the values of a static field or the call to *Return_SIO*. *Var* can be a variable, a dereferencing of the memory or the set of the values of a static field.

QCC: $\forall\ \overline{value} \in \overline{S}$:
  $cond(\overline{value}) \longrightarrow$
  $cstr(\overline{value})$
S: $\mathcal{V}\ [\![x,m,ctx]\!] \mid \mathcal{C}\ [\![m,ctx]\!] \mid Const\ Set \mid$
  $Lookup\ (Sig,\ Id_{ci},\ Id_{ci},\ E_{ci})$
$cond$: $H_1 \wedge \cdots \wedge H_n$
$Condition\ (H)$:
  $CI\_Visibility?\ (Id_c, Id_{ci}) \mid$
  $Method\_Visibility?\ (Id_c, Id_c, Desc_m) \mid$
  $Field\_Visibility?\ (Id_c, Id_f) \mid$
  $AccessInterface?\ (Ref, Ref, Id_i) \mid$
  $AccessPutstatic?\ (Ref, Value) \mid$
  $value \in Const\ Set$
$cstr$: $\mathcal{P}(SC)$
$Constraint\ (SC)$: $Exp \subseteq Var$
Exp: $Const\ Set \mid \mathcal{V}\ [\![x,m,ctx]\!] \mid \mathcal{SF}\ [\![Id_{ci}]\!](Id_f) \mid$
  $\mathcal{C}\ [\![m,ctx]\!] \mid mem(Ref).Fldv(Id_f) \mid$
  $Return\_SIO$ (AID,AID)
Var: $\mathcal{V}\ [\![x,m,ctx]\!] \mid \mathcal{SF}\ [\![Id_{ci}]\!](Id_f) \mid \mathcal{C}\ [\![m,ctx]\!] \mid$
  $mem(Ref).Fldv(Id_f)$

5.2 Analysis

The analysis generates, for each method and for an execution context *ctx*, a set of *QCC*s that describes the data flow of the method in this context. The set of constraints for a method is the union of the set of constraints for each instruction. The function to analyze an instruction is:

$$\mathcal{A}_{Inst}: Inst \times Id_m \times Context \rightarrow \mathcal{P}(QCC)$$

This function takes three parameters: the instruction to analyze, the current method, and the context in which the method is analyzed. An instruction is just a program counter and the bytecode instruction at this address. In the following we define this function for each bytecode instruction.

getstatic The `getstatic` instruction loads a value stored in a static field of a class or interface and stores it into a local variable. The value in the field *C.f* is stored in the local variable T if and only if the field exists and the field is visible at instruction *Inst* (figure 2).

invokeinterface The `invokeinterface` instruction makes a call to an interface method. We calculate the set of methods to which the method signature *sig* can be resolved

$$Lookup\ (sig, mem(o).Type, \lceil p \rceil^{p.c}, E_{ci})$$

together with the context in which the methods called will be analyzed *(Prev,App)*. If the call is accepted by the firewall ($AccessInterface?\ (r, o, \lceil p \rceil^{p.c}, E_{ci})$), we add constraints to simulate this call. We create constraints to simulate the transfer of the actual parameters to the formal parameters:

$$\mathcal{V}\ [\![T_i,m,ctx]\!] \subseteq \mathcal{V}\ [\![P_i,q,(Prev,App)]\!],$$

and add a constraint to retrieve the value returned by the method called

$$\mathcal{V}\ [\![T_0,m,ctx]\!] \supseteq \mathcal{V}\ [\![R,q,(Prev,App)]\!].$$

Finally, we add the object o in $\mathcal{C}\ [\![q,(Prev,App)]\!]$ to indicate that the method q was invoked on this object (figure 3).

load The `load` instruction loads value contained in a variable and stores it in an other variable. The values contained by the variable T_2 are transfered into the variable T_1 (figure 4).

new The `new` instruction simulates the creation of a new class instance and stores a reference to it into a variable. If the class is visible by the instruction, we store in $\mathcal{V}\ [\![T,m,ctx]\!]$ the reference to the created object (figure 5).

putstatic The `putstatic` instruction stores a value in a static field. The value contained in variable T is stored in the static field f of the class $\lceil f \rceil^{p.c}$ if the field is visible by the instruction and if the firewall accepts this access (figure 6).

$$\mathcal{A}_{Inst}\ ((pc, T := \texttt{getstatic}\ f),\ m,\ ctx) = \begin{array}{l} \forall\,(r) \in \mathcal{C}\,[\![m, ctx]\!] : \mathit{Field\_Visibility?}(mem(r).Id_{ci}, f, E_{ci}) \\ \quad \rightarrow \{\mathcal{V}\,[\![T, m, ctx]\!] \supseteq \mathcal{SF}\,[\![\lceil f \rceil^{p.c}]\!](f)\} \end{array}$$

Figure 2: Getstatic

$$\begin{array}{l} \mathcal{A}_{Inst}\ ((pc, T_0 := \texttt{invokeinterface}\ p\ T_1\ T_2\ \cdots\ T_n\ S_2 :: \cdots :: S_n :: S_{n+1}),\ m,\ ctx) = \\ \qquad \forall\,(r, o, q) \in \mathcal{C}\,[\![m, ctx]\!] \times \mathcal{V}\,[\![T_1, m, ctx]\!] \times \mathit{Lookup}(sig, mem(o).Type, \lceil p \rceil^{p.c}, E_{ci}) \\ \qquad\qquad : \mathit{AccessInterface?}(r, o, \lceil p \rceil^{p.c}, E_{ci}) \\ \qquad\qquad \rightarrow \left\{ \begin{array}{l} \mathcal{V}\,[\![T_1, m, ctx]\!] \subseteq \mathcal{V}\,[\![P_1, q, ctx']\!], \\ \cdots \\ \mathcal{V}\,[\![T_n, m, ctx]\!] \subseteq \mathcal{V}\,[\![P_n, q, ctx']\!], \\ \mathit{Init\_Var}(E_{ci}(\lceil q \rceil^{p.c}).Mtd((\lceil q \rceil^{m}, S_2 :: \cdots :: S_{n+1})).Varl, q, ctx') \\ \mathcal{C}\,[\![q, ctx']\!] \supseteq \{o\} \\ \mathcal{V}\,[\![T_0, m, ctx]\!] \supseteq \mathcal{V}\,[\![R, q, ctx']\!] \end{array} \right\} \end{array}$$

where we have used the following abbreviations:

$$\begin{aligned} sig &= (\lceil p \rceil^{m}, S_2 :: \cdots :: S_n) \\ P_1 :: \cdots :: P_n &= (E_{ci}(\lceil q \rceil^{p.c}).Mtd)((q, S_2 :: \cdots :: S_n)).Param \\ ctx' &= (Prev, App) \\ App &= (mem(r).Owner.Id_p, mem(r).Owner) \\ Prev &= \begin{cases} ctx.Prev \text{ if } ctx.App.Id_p = App.Id_p \\ ctx.App \text{ otherwise} \end{cases} \\ R &= (E_{ci}(\lceil q \rceil^{p.c}).Mtd)((q, S_2 :: \cdots :: S_n)).Res.Id_v \end{aligned}$$

Figure 3: Invokeinterface

$$\mathcal{A}_{Inst}\ ((pc, T_1 := \texttt{load}\ T_2),\ m,\ ctx) = \ \rightarrow \{\mathcal{V}\,[\![T_1, m, ctx]\!] \supseteq \mathcal{V}\,[\![T_2, m, ctx]\!]\}$$

Figure 4: Load

$$\mathcal{A}_{Inst}\ ((pc, T := \texttt{new}\ c),\ m,\ ctx) = \begin{array}{l} \forall\,(r) \in \mathcal{C}\,[\![m, ctx]\!] : \mathit{CI\_Visibility?}(mem(r).Id_{ci}, c, E_{ci}) \\ \quad \rightarrow \{\mathcal{V}\,[\![T, m, ctx]\!] \supseteq \{(pc, r.Owner)\}\} \end{array}$$

Figure 5: New

$$\begin{array}{l} \mathcal{A}_{Inst}\ ((pc, \texttt{putstatic}\ f\ T),\ m,\ ctx) = \\ \qquad \forall\,(r, v) \in \mathcal{C}\,[\![m, ctx]\!] \times \mathcal{V}\,[\![T, m, ctx]\!] \\ \qquad\quad : \mathit{Field\_Visibility?}(mem(r).Id_{ci}, f, E_{ci}) \wedge \mathit{AccessPutstatic?}(r, v) \\ \qquad \rightarrow \{\mathcal{SF}\,[\![\lceil f \rceil^{p.c}]\!](f) \supseteq \{v\}\} \end{array}$$

Figure 6: Putstatic

$$\mathcal{A}_{Inst}\ ((pc, T_1 := \texttt{store}\ T_2),\ m,\ ctx) = \ \rightarrow \{\mathcal{V}\,[\![T_1, m, ctx]\!] \supseteq \mathcal{V}\,[\![T_2, m, ctx]\!]\}$$

Figure 7: Store

$$\begin{array}{l} \mathcal{A}_{Inst}\ ((pc, T := \texttt{invokestatic}\ \mathit{getPrevCtx}),\ m,\ ctx) = \\ \qquad \left\{ \begin{array}{l} \forall\,(r) \in \mathcal{C}\,[\![m, ctx]\!] : ctx.App.Id_p = mem(r).Owner.Id_p \\ \quad \rightarrow \{\mathcal{V}\,[\![T, m, ctx]\!] \supseteq ctx.Prev.AID\} \\ \\ \forall\,(r) \in \mathcal{C}\,[\![m, ctx]\!] : ctx.App.Id_p \neq mem(r).Owner.Id_p \\ \quad \rightarrow \{\mathcal{V}\,[\![T, m, ctx]\!] \supseteq ctx.App.AID\} \end{array} \right\} \end{array}$$

Figure 8: getPrevCtx

Let $\mathcal{A}_{Inst}\ ((pc, BCinst),\ m,\ ctx) = \forall\,\bar{v} \in \bar{E} : cond \rightarrow \{C\}$. Then

$$\begin{array}{l} \mathcal{A}_{Inst}\ ((pc, \texttt{ifAID}\ T \in S\ BCinst),\ m,\ ctx) = \\ \qquad \forall\,(\bar{v}, a) \in \bar{E} \times \mathcal{V}\,[\![T, m, ctx]\!] : cond \wedge a \in S \rightarrow \{C\} \end{array}$$

Figure 9: ifAID

Figure 10: Examples of *QCCs*

store The `store` instruction stores the value contained in variable T_2 in variable T_1. This data flow is modeled by a simple set inclusion: values contained in variable T_2 may also be contained in variable T_1 (figure 7).

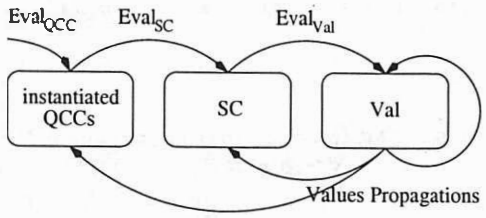
getPrevCtx The instruction `invokestatic` *getPrevCtx* makes a call on the static method *JCSystem.getPreviousContextAID*. The method *getPrecCtx* serves to find the AID of the active applet before the last context switch. The first constraint is activated when the active context is the context of the caller, in which case they have the same previous context. The second one is activated when the active context differs from the context of the caller. In that case the previous context is the context of the caller (figure 8).

ifAID The *QCC* used in this construct is the one analyzed for the *BCinst* instruction. A condition is added such that the constraints are only generated if the condition in the test is true (figure 9).

6 Resolution

The resolution of quantified conditional constraints can be done iteratively as an ordinary fix point computation. The main difference with a "classic" system is that the set of constraints and the values of variables in the constraints evolve together. Hence, the iteration sequence consists of triples (qcc,sc,val) where qcc is the current set of quantified conditional constraints instantiated for particular contexts, sc is the current set of simple constraints and val is a valuation that to each variable associates its current value.

Suppose that we have a program P consisting of a set of applets (*Aplt*) and a set of methods (*Meth*). Let Q be the set of (uninstantiated) *QCC*s obtained by analyzing P (with functions $\mathcal{A}_{Class}$ for a class or an interface, $\mathcal{A}_{Meth}$ for a method and $\mathcal{A}_{Inst}$ for an instruction). During the resolution of Q, we compute the new set of instantiated *QCC*s, $\mathcal{P}(QCC)$, with the function $Eval_{QCC}$, the new set of simple constraints *SC*, $\mathcal{P}(SC)$, with the function $Eval_{SC}$ and the new valuation *Val* with the function $Eval_{Val}$, as defined below.



The function $Eval_{QCC}$ uses the current valuation to instantiate the *QCC*s in the set Q and adds the corresponding constraints to the current set of constraints. This is where the resolution becomes context-sensitive: if a method is not called in a particular context, no constraints for this method will be generated in that particular context.

$$
\begin{aligned}
&Eval_{QCC}:\\
&\quad \mathcal{P}(QCC) \times Val \longrightarrow \mathcal{P}(QCC)\\
&Eval_{QCC}\,(qcc,val) =\\
&\quad qcc \cup \bigcup_{\substack{m \in Meth\\ ctx \in Context}} \left\{ ctr \;\middle|\; \begin{array}{l} o \in \mathcal{C}\,[\![m,ctx]\!]\\ \wedge\, ctx' = CalcCtx\,(o,ctx,val)\\ \wedge\, ctr \in \mathcal{A}_{Meth}\,(m,ctx') \end{array} \right\}
\end{aligned}
$$

where the function for calculating the context of the call is given by

$$
\begin{aligned}
&CalcCtx:\\
&\quad Ref \times Context \times Val \longrightarrow Context\\
&CalcCtx\,(r,c,v) = (Prev,\ App)\\
&\quad \textbf{where}\\
&App = (v(mem))(r).Owner\\
&Prev = \begin{cases} c.Prev \ \textbf{if}\ c.App.Id_p = App.Id_p\\ c.App\ \textbf{otherwise} \end{cases}
\end{aligned}
$$

The function $Eval_{SC}$ uses the current valuation to verify the condition for each constraint in the set of instantiated *QCC*s and adds the corresponding simple constraints to the current set of constraints. This evaluation permits to restrict the production of the simple constraints that model the effect of an instruction that "executed". We use the notation $[\![\text{Exp}]\!]_V$ to denote the evaluation of the expression *Exp* with the values contained by the valuation V.

$$
\begin{aligned}
&Eval_{SC}:\\
&\quad \mathcal{P}(QCC) \times \mathcal{P}(SC) \times Val \longrightarrow \mathcal{P}(SC)\\
&Eval_{SC}\,(qcc,sc,val) =\\
&\quad sc \cup \left\{ ctr[\overline{v}/\overline{x}] \;\middle|\; \begin{array}{l} \forall' \overline{x} \in \overline{X} : cond \longrightarrow ctr \in qcc\\ \wedge\, \overline{v} \in [\![\overline{X}]\!]_{val}\\ \wedge\, \overline{cond[v/x]} \end{array} \right\}
\end{aligned}
$$

The function $Eval_{Val}$ is the standard evaluation function associated to a constraint set. For every constraint $exp \subseteq var$ in the current constraint set *cs* we evaluate the expression with the current valuation and add the new value in *val(var)*.

```
$Eval_{Val}$:
    $\mathcal{P}(SC) \times Val \rightarrow Val$
$Eval_{Val}(sc, val) =$
    $val[var \mapsto val(var) \sqcup [\![exp]\!]_{val}]$
        with
    $exp \subseteq var \in sc$
```

```
ALGORITHM
$Q := \bigcup_{A \in Aplt} \mathcal{A}_{Class}(A)$ ;
$qcc' := \mathcal{A}_{Class}(JCRE)_{(JCRE,JCRE)}$ ;
$sc, sc', qcc := \emptyset$ ;
$val := \bot$ ;
$val' := val_0$[8] ;
while $qcc \neq qcc'$ or $sc \neq sc'$ or $val \neq val'$ do
    $qcc := qcc'$ ; $sc := sc'$ ; $val := val'$ ;
    $qcc' := Eval_{QCC}(qcc, val)$ ;
    $sc' := Eval_{SC}(qcc, sc, val)$ ;
    $val' := Eval_{Val}(sc, val)$ ;
endwhile
END
```

Proposition 6.1 *This algorithm terminates with a correct solution to Q.*

The proof of Proposition 6.1 is an extension of the standard argument based on Tarski's theorem [24, 11]. The specificity of the proof is to take into account that the system evolves (in a monotonic fashion!) during the computation. The formal proof (termination and correctness) can be found in [16].

Establishing a start state for the iteration requires special attention in Java Card because there is no `main` to initialize the analysis. The sequence of operations is given by the JCRE and the user. We model this interaction with the card by adding an artificial JCRE applet that is analyzed like the others. For the JCRE we know its context (it is (JCRE,JCRE)) which permits the algorithm to produce the initial set of instantiated *QCCs*. The initial valuation val_0 links each element with its default value. For each $\mathcal{V}[\![x,m,ctx]\!]$ and $\mathcal{C}[\![m,ctx]\!]$ the default value is $\emptyset$. For each $\mathcal{SF}[\![Id_{ci}]\!]$ the default value is the function which links each static field of Id_{ci} with its default value ($\emptyset$ for a reference and $\{P\}$ for a primitive *P*). Finally, we initialize the abstract memory (*mem*) with the undefined abstract objects for each abstract reference.

[8]The definition of the initial value val_0 comes after the algorithm.

7 An example analysis

In figure 11, we present a variation of the example given in section 2.1, in which the firewall and Alice can not prevent the flow of the Alice secret to Charlie. Here, Bob implements a shareable object and passes a reference to it to Charlie. In this case, the invoke at *Alice.foo* is valid at runtime, because for Alice the caller is always Bob. Here, we only present the transformation of this example in our language in the figure 12. The constraints are neither generated nor solved automatically yet, but we work on an implementation of the previously presented algorithm. During the resolution, each "variable" received the possible values that it can contain. In this example, the important value is the secret of Alice (represented by the reference $(p, AliceAID)$) and the important variable is the static field `AliceSecret` of Charlie. The resolution gives, as a part of the global solution, the following possible value for the static field of Charlie:

$$(\mathbf{p}, \mathbf{AliceAID}) \in \mathcal{SF}[\![\mathbf{Charlie}]\!](\mathbf{Charlie.AliceSecret})$$

This result proves that there is an illegal object flow with the secret of Alice.

8 Related works

The formalization of the Java Card firewall has been the object of several works. Motré [19] has formalized the firewall with the B method. She defines a machine for the firewall and an operation for each check of the firewall. This modeling provides a formal description of the firewall that is used to ensure that the firewall verifications are sufficient to fulfill the security policy. In addition, successive refinements lead to a reference implementation of the firewall. More traditional operational semantics for modeling the firewall checks have been given by Éluard *et al.* [17]. Siveroni *et al.* [22] show how to integrate this into an operational semantics for Java Card. For the modeling of the JCRE it is necessary to be able to "execute" the differents applets. We choose to follow the approach used by Attali *et al.* [3, 4] and model the JCRE by an applet. With this approach, we can adapt the JCRE to obtain either exactly the execution we want or all possible executions.

The problems related to the Java Card fire-

```
public class Bob extends Applet
      implements MSI2{
   private static MSI AliceObj;
private void bar () {
   AliceObj=(MSI) getSIO (AliceAID); }
public Secret foo2 () {
   return AliceObj.foo (); } }
```

```
public class Charlie extends Applet {
   private static MSI2 BobObj;
   private static Secret AliceSecret;
private void bar () {
   BobObj=(MSI2) getSIO (BobAID); }
private void foo3 () {
   AliceSecret=BobObj.foo2 (); } }
```

Figure 11: An example of illegal object flow

```
public class Alice extends Applet implements MSI {
   private Secret Object.Secret;
   public Secret foo () {
      AID Client;
      Secret Response;
      1:T1:=invokestatic getPrevCxt
      2:Client:=store T1
      3:ifAID Client ∈ {BobAID} T2:=getstatic Alice.ObjectSecret
      4:Response:=store T2
      5:Alice.foo_Ret:=load Response
      return Alice.foo_Ret } }
```

```
public class Bob extends Applet
      implements MSI2{
   private static MSI AliceObj;
public Secret foo2 () {
   6:T3:=getstatic Bob.AliceObj
   7:T4:=invokeinterface MSI.foo T3
   8:Bob.foo2_Ret:=store T4
   return Bob.foo2_Ret } }
```

```
public class Charlie extends Applet {
   private static MSI2 BobObj;
   private static Secret AliceSecret;
private void foo3 () {
   9:T5:=getstatic Charlie.BobObj
   10:T6:=invokeinterface MSI2.foo2 T5
   11:putstatic Charlie.AliceSecret T6} }
```

Figure 12: The translation of the three methods of the example in our language

wall have been observed by others, notably Montgomery and Krishna [18], who propose another approach to secure object sharing based on delegates. A server implements a delegate object that mediates access to those methods that the server wants to share with others. The delegate object performs the checks that it deems necessary to grant access. This approach is more flexible than the existing firewall but has the drawback that it requires (minor) changes to the JCVM. This technique permits to use more sophisticated authentication mechanisms than the one based only on AID comparison. In the paper it is shown how to use a protocol based on challenge/response phrases to avoid the problem of AID spoofing. However, no technique is presented for proving that delegates indeed do respect a given security policy. In contrast, our approach works for the standard JCVM and relies on static analysis to check that no unwanted access takes place.

Two works on the verification of applet sharing on Java Card are closely related to ours. Bieber *et al.* [8, 7], as part of the Pacap project [2], have defined an analysis of Java Card applets which can detect illegal information flow. Their approach is based on three elements: an abstraction of values of variables into a *level* that describes the sharing of the value, an invariant that is a sufficient condition the security property to hold and a model checker to verify the invariant. A lattice of levels is used to represent the sharing of objects. If an applet A is allowed to share some information with an applet B, the level $A+B$ is entered into the lattice specifying the security policy. Each applet is represented by a call graph and each call graph is transformed into an SMV model. To work with a shareable object, an applet must call an interface method so only call graphs which include an interface method are taken into account. The invariant together with the control flow graphs are given to the SMV model checker for verification. The work presented here complements their work by providing a precise description of how these control and data flow graphs can be calculated, taking into account the firewall and the different calling contexts.

The analysis proposed by Caromel, Henrio and Serpette [9] has as aim to signal whether a security exception might (or will definitely) be raised by the firewall at execution of a set of applets. The analysis thus shares objectives with ours and calculates the same type of information. The differences between the analyses lie in the precision. Caromel *et al.* have opted for a simple, flow-insensitive analysis whereas we can obtain some flow sensitivity through the choice of local variables in our three-address byte code. Instead of modeling the memory state explicitly, they use an alias analysis to track side effects of assignments. The control flow analysis in their analysis is a simple class hierarchy analysis, in contrast to our context-sensitive flow analysis. Indeed, their analysis does not analyze methods separately for each calling context and hence would not be able to deal with the call stack inspection as well as our analysis. Thus, the two analyses can be seen as two extremes of the design space for flow analysis for Java Card.

The quantified conditional constraints (*QCC*s) introduced in Section 5.1 are an extension of the conditional constraints (originally due to Reynolds [21]) that are used in the object-oriented type analysis defined by Palsberg and Schwartzbach [20]. In this analysis, conditions of the form $C \in \mathcal{V}(X)$ are used to guard the constraints generated from class C such that these are only evaluated when class C is actually used. However, it is still necessary to generate the constraints for every class in the hierarchy which leads to scalability problems. The *QCC*s, on the other hand, generate these constraints *on demand*: only when the analysis discovers that a certain class or method is used, the corresponding constraints are generated and added to the current set of constraints.

9 Conclusions and future work

The access control exercised by the Java Card firewall is bypassed when invoking methods on shareable objects. In order to determine the access control that is implemented by a given set of Java Card applets we have presented a static analysis that calculates a safe approximation of the flow of objects between applets of a Java Card application. The static analysis is an extension of the constraint-based program analysis framework that allows to generate and solve data flow constraints in a demand-driven fashion.

The information calculated by our analysis has other applications than verifying access control. The data flow information allows to construct a precise *control flow graph* on which other safety-style properties of the application can be verified. Examples of these include verifying that all Java Card transactions are well-formed and that exceptions are properly caught and treated by the application. A verification technique based on model checking using finite automata is detailed in [16].

The present analysis does not deal with the problem of *(indirect) information flow* between applets. In particular, we do not model the flow of primitive values between applets so we cannot detect if applet B transfers data to applet C that contains information obtained from applet A. Analyses for detecting such information flow have been proposed elsewhere (see *e.g.* [25]) in the setting of a simple imperative language. The control and object flow information calculated by our analysis can be used to adapt such analyses to the Java Card language because it allows to eliminate the higher-order and object-oriented features of an application, essentially translating it into an imperative language. This requires an improvement to the abstract domains such that owner information can be attached to primitive values and primitive operations must be adjusted to calculate the possible owners depending on the values used in the operation as well as the applet which does the operation.

Finally, for the moment the analysis does not take into account exceptions other than security exceptions. With the current abstraction of the primitive values it is clear that exceptions related to *e.g.*, array access (index-out-of-bound exceptions) can only be dealt with in a very approximate fashion. Exceptions form an integral part of the control-flow of an application so progress in this direction is desirable.

References

[1] Java Card 2.1.1. `http://java.sun.com/products/javacard/javacard21.html`, 2001.

[2] The PACAP project. `http://www.gemplus.com/smart/r_d/projects/pacap.htm`, 2001.

[3] Isabelle Attali, Denis Caromel, Carine Courbis, Ludovic Henrio, and Henrik Nilsson. Smart tools for Java Cards. In Josep Domingo-Ferrer, David Chan, and Anthony Watson, editors, *CARDIS*. Kluwer Academic Publishers, September 2000.

[4] Isabelle Attali, Denis Caromel, Carine Courbis, Ludovic Henrio, and Henrik Nilsson. An integrated development environment for Java Card. *Special issue on Smart Cards of the Journal Computer Networks*, 36(4):391-405, July 2001.

[5] Peter Bertelsen. Semantics of Java Byte Code. Technical report, DTU, March 1997. Home page `http://www.dina.kvl.dk/~pmb/`, 2001.

[6] Peter Bertelsen. Dynamic semantics of Java bytecode. In *Workshop on Principles on Abstract Machines*, September 1998. Home page `http://www.dina.kvl.dk/~pmb/`, 2001.

[7] Pierre Bieber, Jacques Cazin, Pierre Girard, Jean-Louis Lanet, Virginie Wiels, and Guy Zanon. Checking secure interactions of smart cards applets. In *ESORICS*, volume 1895 of *LNCS*, pages 1–16. Springer-Verlag, 2000.

[8] Pierre Bieber, Jacques Cazin, Abdellah El Marouani, Pierre Girard, Jean-Louis Lanet, Virginie Wiels, and Guy Zanon. The PACAP prototype : a tool for detecting Java Card illegal flow. In Isabelle Attali and Thomas Jensen, editors, *JCW*, volume 2041 of *LNCS*, September 2000.

[9] Denis Caromel, Ludovic Henrio, and Bernard Serpette. Context inference for static analysis of java card object sharing. In Isabelle Attali and Thomas Jensen, editors, *ESMART*, volume 2140, September 2001.

[10] Zhiqun Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide.* Addison-Wesley, 2000.

[11] Patrick Cousot and Radhia Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *FPCA*, pages 170–181. ACM Press, June 1995.

[12] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Second Edition.* Addison-Wesley, 2000. 896 pages, `http://java.sun.com/docs/books/jls/index.html`, 2001.

[13] Paul R. Hudak. A semantic model of reference counting and its abstraction. In Samson Abramsky and Chris Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 3, pages 45-62. Ellis Horwood series in computer and their applications, 1987.

[14] Thomas Jensen, Daniel Le Métayer, and Tommy Thom. Security and dynamic class loading in Java: A formalisation. In *ICCL*, pages 4–15, May 1998.

[15] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification, Second Edition.* Addison-Wesley, 1999. `http://java.sun.com/docs/books/vmspec/index.html`, 2001.

[16] Marc Éluard. *Analyse de sécurité pour la certification d'applications Java Card.* PhD thesis, Université de Rennes 1, December 2001. N. d'ordre : 2614.

[17] Marc Éluard, Thomas Jensen, and Ewen Denney. An operational semantics of the Java Card firewall. *LNCS*, 2140:95–110, September 2001.

[18] Michael Montgomery and Ksheerabdhi Krishna. Secure object sharing in Java Card. In *Smartcard*, pages 119–127. USENIX, May 1999.

[19] Stéphanie Motré. Modélisation et implémentation formelle de la politique de sécurité dynamique de la Java Card. In *AFADL*, pages 158–172. LSR/IMAG, January 2000.

[20] Jens Palsberg and Michael I. Schwartzbach. *Object-Oriented Type Systems.* John Wiley & Sons, 1994.

[21] John C. Reynolds. Automatic computation of data set definitions. In *Information Processing*, volume 1, pages 456-461. North-Holland, August 1969.

[22] Igor Siveroni, Thomas Jensen, and Marc Éluard. A formal specification of the Java Card applet firewall. In Hanne Riis Nielson, editor, *NordSec*, Technical Report IMM-TR-2001-14, pages 108–122. Technical University of Denmark, November 2001.

[23] Soot: a Java optimization framework. `http://www.sable.mcgill.ca/soot/`, 2001.

[24] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, June 1955.

[25] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *JCS*, 4(3):167–187, December 1996.

Extending Tamper-Proof Hardware Security to Untrusted Execution Environments

Sergio Loureiro, Laurent Bussard, and Yves Roudier

Institut Eurecom, 2229 route des Crêtes, B.P. 193
06904 Sophia Antipolis, France
{loureiro, bussard, roudier}@eurecom.fr

Abstract

This paper addresses mobile code protection with respect to potential integrity and confidentiality violations originating from the untrusted runtime environment where the code execution takes place. Both security properties are defined in a framework where code is modeled using Boolean circuits. Two protection schemes are presented. The first scheme addresses the protection of a function that is evaluated by an untrusted environment and yields an encrypted result only meaningful for the party providing the function. The second scheme addresses the protection of a piece of software executed by an untrusted environment. It enforces the secure execution of a series of functions while allowing interactions with the untrusted party. The latter technique relies on trusted tamper-proof hardware with limited capability. Executing a small part of the computations in the tamper-proof hardware extends its intrinsic security to the overall environment.

1 Introduction

The mobile code paradigm is becoming increasingly praised for its flexibility in the management of remote computers and programmable devices. Unsurprisingly, more flexibility leads to new challenging security problems. Mobile code presents vulnerabilities unheard of in the traditional programming world. On one hand, attacks may be performed by mobile programs against a remote execution environment and its resources. On the other hand, a mobile code may be subverted by a malicious remote execution environment. The former issue has been widely addressed [25], for instance through containment mechanisms like the sandbox, the applet-firewall, etc., but few solutions deal with the latter.

This paper extends our work on the protection of mobile code [23], [24]. The problem addressed here is as follows: Alice (A) wants a piece of code to be executed on Bob's (B) workstation, the result of its execution being eventually returned to A. However, B cannot be trusted and might try to modify the execution of this program. In addition, in the context of mobile code, interacting with A during the program execution is not an option. In other words, the code sent by A must be executed autonomously by B who only provides additional input parameters. It is necessary to ensure that B cannot get information about the semantics of the code provided by A and that A can be assured, without performing the computation herself, that the execution has not been tampered with. This is different from volunteer computing scenarios [34] such as Seti@home where data to treat is provided by A.

In this model, "integrity of execution" means that B cannot alter the execution of the program and surreptitiously modify its results. "Confidentiality of execution", sometimes termed "privacy of computation" although it bears no relationship with anonymity, aims at preventing the disclosure of program semantics. Integrity and confidentiality are tightly entangled in our proposal because of the cryptographic protection scheme used. However, confidentiality and integrity are independent properties and thus will be evaluated separately in each solution.

The use of tamper-proof hardware (TPH) has long been the only solution for protecting the execution of critical programs from an untrusted party: the program is completely executed within the hardware that is trusted by the code owner. With the advent of mobile code, TPH has logically been advocated as the most obvious solution for protecting a program from its untrusted execution environment. [43] is a good example of this hardware-only trend. However, existing solutions based on tamper-proof hardware suffer from inherent limitations ranging from the cost and difficulty of retrofitting tamper-proof and powerful cryptographic boards on everybody's workstations to the lack of computing power in smart cards.

Prompted by the limitation of TPH-based solutions, alternative approaches were brought up as application-specific solutions [10], solutions aimed at protecting specific classes of mathematical functions [33], [32], or even empirical and mathematically unfounded ones like obfuscation [16]. Our proposal also takes into account the inherent limitations of tamper-proof hardware. Mathematical functions, which can be represented by Boolean circuits, are a building block for programs. Section 2 presents a scheme ensuring a secure non-interactive evaluation of such functions. To this end, the circuit implementing the function is encrypted using a technique inspired by the McEliece public key scheme [28]. Based on this solution, Section 3 describes a scheme for the secure non-interactive evaluation of a piece of software consisting of the combination of several functions and of a control structure scheduling these functions. In this case, a Tamper-Proof Hardware acting on behalf of the party providing the mobile code is required. Executing a small part of the computations in the TPH extends the intrinsic security of the TPH to the overall environment. In Section 4, this solution is compared with similar approaches dealing with integrity or confidentiality of execution.

2 Protecting Functions

As a first step towards integrity and confidentiality of execution, a solution for protecting mathematical functions is proposed. This solution is inspired by the work of Sander and Tschudin [33], [32], who devised a function hiding scheme for non-interactive protocols (see Section 4.1).

Figure 1 describes the main steps of the function protection process using this solution. Using E_A, function f is encrypted by its originator A into a new function f'. The untrusted host B evaluates f' on the cleartext input x and gets y' as the encrypted result of this evaluation. Using the secret decryption algorithm D_A, A can retrieve y that is the cleartext result of the original function f, based on the following property of the function hiding scheme: $y = D_A(y') = D_A(f'(x)) = D_A(E_A(f(x))) = f(x)$. Moreover, an integrity verification algorithm V_A is used by A to check the computation performed by B.

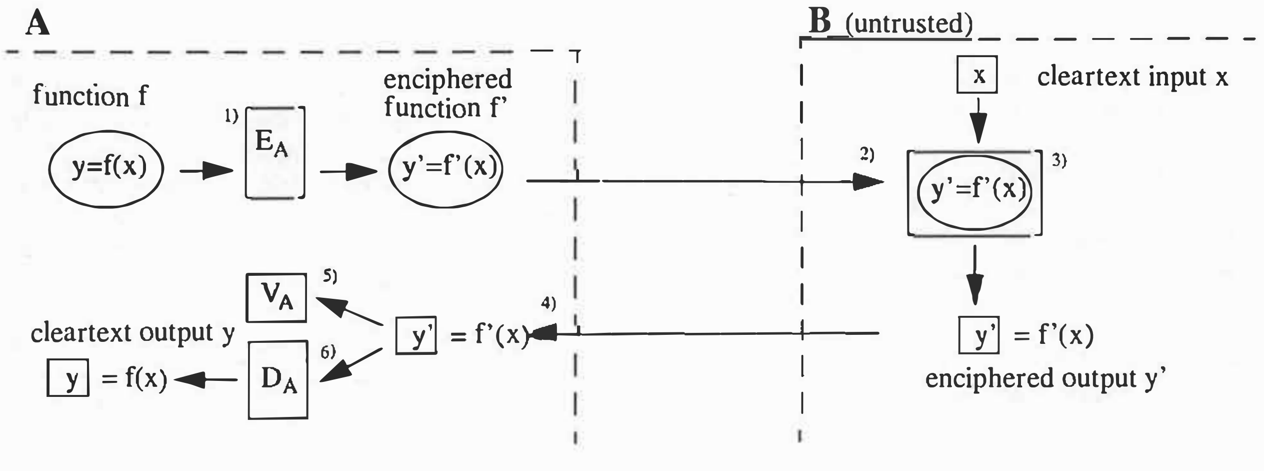


Figure 1: Evaluating an encrypted function on an untrusted host.
1) the function is encrypted; 2) the encrypted function is sent to the untrusted host (confidentiality); 3) the encrypted function is evaluated with cleartext inputs; 4) the encrypted result is sent back to *A*; 5) the result is verified (integrity); 6) Decipherment is performed to obtain the cleartext result.

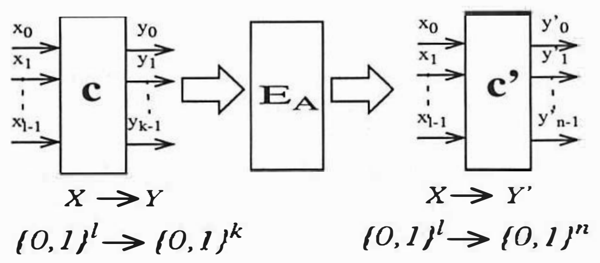


Figure 2: Construction of the encrypted circuit c'.

2.1 Computational Model

This section defines the mechanisms required to ensure integrity and confidentiality of execution in an untrusted environment. Section 2.2 describes more precisely how those concepts are implemented in our approach.

General Overview

Since fixed-length inputs and outputs are used, it is possible to deal with functions using a Boolean circuit representation. Let us represent the function *f* with a circuit called *c* (in the sequel of this paper, circuit and function are largely used as equivalents). The number of binary inputs (*l*) and outputs (*k*) will be defined according to the possible input and output values of the function. *X* is the unrestricted set of all possible inputs (e.g. $\{0,1\}^l$). $F_{l,k}$ represent the family of Boolean circuits with *l* inputs and *k* outputs ($X \rightarrow Y$). Circuit $c \in F_{l,k}$ defines a relation between input $x \in X$ and output $y = \{c(x_i) | (x_i \in X)\}$, $y \in Y$, $Y \subseteq \{0, 1\}^k$. The circuit *c* may also be seen as a set of *k* functions: $\{0, 1\}^l \rightarrow \{0, 1\}$. Each of these functions is defined by a Boolean equation. The corresponding *k* equations are the inputs to algorithm E_A (Figure 2). The result is a set of *n* Boolean equations that define a new Boolean circuit $c' \in F_{l,n}$. The circuit *c'* defined by *A* is evaluated by *B* who provides input data $x \in X$, but the encryption by E_A prevents the disclosure of *c* to *B*. Look at Section 2.2 for details on the encryption mechanism E_A.

Confidentiality of Execution

The circuit $c'=E_A(c)$ preserves the confidentiality of *c* if it is computationally infeasible to derive *c* from *c'*. A decryption algorithm D_A must be used in order to retrieve the desired cleartext result $y=c(x)$ from the obtained ciphertext result $y'=c'(x)$. A polynomial time decryption algorithm is necessary to remain realistic.

Integrity of Execution

Alice receives a ciphertext result *y'* corresponding to the evaluation of the encrypted circuit *c'* with *Bob*'s cleartext inputs. *Alice* should be able to retrieve from *y'* the cleartext result *y* corresponding to the evaluation of the circuit *c* using the same input. V_A has to define a polynomial time verification of this result since for practical applications, the circuit owner must be able to efficiently verify the result of the circuit execution. The verifier concept is introduced to address the problem of integrity of execution. The verifier shares some similarities with CS Proofs [29] in that there can exist invalid proofs but those should be hard to find. Basically, the verifier concept relies on the difficulty of finding valid values (*y'*) that do not correspond to valid cleartext outputs (*y*). Using the terminology of [9], *Alice*'s verifier V_A checks that there exists an *x* such that $y=c(x)$. It can be defined as follows:

$$\text{if } y \notin \{c(x_i) | (x_i \in X)\} \text{ then } p(V_A(y') = Accept) < \delta$$

Even if the result is verified and cannot be forged randomly, a malicious remote host is able to identify possible outputs of a circuit for chosen inputs. Therefore, integrity of execution alone (i.e. without confidentiality of execution) does not prevent *B* from performing several executions of the circuit and selecting the best result. A scheme that ensures both integrity and confidentiality of execution is thus highly desirable.

2.2 Detailed Protection Scheme

This section presents our solution to encrypt functions. A technique derived from the McEliece [28] public key cryptosystem is used. Unlike the McEliece scheme that encrypts data, our approach encrypts functions. Moreover, this asymmetric scheme is used as a symmetric one by keeping both public and private keys secret. As a result, part of the attacks possible against the McEliece cryptosystem are not relevant in our scheme because attackers do not know the public key.

Circuit Encryption

All Boolean equations of the original plaintext circuit *c* are encrypted using the McEliece technique [28] where data are replaced by equations $c' = E_A(c)$:

$$\underbrace{\left[y'_0 \cdots y'_{n-1}\right]}_{c'} = \underbrace{\left[y_0 \cdots y_{k-1}\right]}_{c} SGP + \underbrace{\left[z_0 \cdots z_{n-1}\right]}_{z}$$

Boolean equations $y_i = f_i(x_0..x_{l-1})$ are multiplied by the $k \times n$ matrix SGP (for more details on Boolean circuit encryption, look at Figure 3). G is a generating matrix for a $[n,k,t]$ Goppa code C [27] and t is the number of errors that the code is able to correct. P is a random $n \times n$ permutation matrix. Because of the importance of hiding the systematic form of the code [13], an additional matrix S is used. S is a random dense $k \times k$ non-singular matrix. S, G and P are kept secret by Alice. The SGP matrix multiplication leads to a linear composition of each cleartext Boolean equation of c. The difference with respect to the original McEliece scheme is that the result of the encryption is interpreted as Boolean equations defining the encrypted circuit.

In addition to the SGP multiplication, algorithm E_A introduces errors in the circuit in order to detect integrity attacks and prevent confidentiality attacks as explained below. As error-correcting codes, Goppa codes allow to efficiently remove these errors at decoding time. Errors introduced by E_A can be viewed as an error circuit that, given an l-bit argument, returns an n-bit string with a Hamming weight of t that is computationally indistinguishable from a random n-bit vector with the same weight. Such a function, called z, computationally indistinguishable from the set of functions satisfying the weight restriction exists. [31] proposes an efficient construction for functions that output words of a given weight.

Encrypted Circuit Evaluation and Verification

Once c' is created, the protocol between Alice and Bob is the following:

- Alice sends c' to Bob.
- Bob evaluates c' on his data $x \in X$ and gets the result $y' = c'(x) \in Y'$. There is an increase in the number of Boolean outputs while the number of inputs is kept unchanged. The result y' is then sent to Alice.

① $\mathbf{y} = \mathbf{f}(\mathbf{x})$

$\Downarrow$

$\mathbf{y} = \mathbf{c}(\mathbf{x})$

②
$$\begin{aligned} y_0 &= \bar{x}_0\bar{x}_1\bar{x}_2 + x_1x_2 \\ y_1 &= \bar{x}_0\bar{x}_1x_2 + x_0\bar{x}_1\bar{x}_2 + x_0x_1x_2 \\ y_2 &= x_0 \\ y_3 &= \bar{x}_0\bar{x}_1x_2 + \bar{x}_0x_1x_2 + x_0x_1\bar{x}_2 \end{aligned}$$

cleartext function truth table $\Downarrow$

$$\underbrace{\begin{array}{ccc} x_0 & x_1 & x_2 \\ 0&0&0\\0&0&1\\0&1&0\\0&1&1\\1&0&0\\1&0&1\\1&1&0\\1&1&1 \end{array}}_{X} \underbrace{\begin{array}{cccc} y_0 & y_1 & y_2 & y_3 \\ \hline 1&0&0&0\\0&1&0&1\\0&0&0&0\\1&0&0&1\\0&1&1&0\\0&0&1&0\\0&0&1&1\\1&1&1&0 \end{array}}_{Y} ③ \bullet \overset{GP}{\begin{bmatrix} 1&0&0&1&1&0&0\\0&0&1&1&0&1&0\\1&0&0&1&0&1&1\\1&1&0&0&0&1&0 \end{bmatrix}} ④ = \underbrace{\begin{array}{ccc} x_0 & x_1 & x_2 \\ 0&0&0\\0&0&1\\0&1&0\\0&1&1\\1&0&0\\1&0&1\\1&1&0\\1&1&1 \end{array}}_{X} \underbrace{\begin{array}{ccccccc} y_{GP,0} & y_{GP,1} & y_{GP,2} & y_{GP,3} & y_{GP,4} & y_{GP,5} & y_{GP,6} \\ \hline 1&0&0&1&1&0&0\\1&1&1&1&0&0&0\\0&0&0&0&0&0&0\\0&1&0&1&1&1&0\\1&0&1&0&0&0&1\\1&0&0&1&0&1&1\\0&1&0&1&0&0&1\\0&0&1&1&1&0&1 \end{array}}_{Y_{GP}} ⑤$$

$$Y_{GP} = Y \cdot GP$$

partially encrypted function truth table $\Uparrow$

⑥
$$y_{GP,0} = \bar{x}_0\bar{x}_1 + x_0\bar{x}_1$$
$$y_{GP,1} = \dots ; \quad \dots \quad ; y_{GP,6} = \dots$$

$\mathbf{y}_{\mathbf{GP}} = \mathbf{c}_{\mathbf{GP}}(\mathbf{x})$

$||$

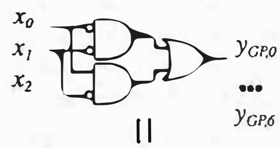
⑦ 

Figure 3: Encrypting a circuit : basic steps
For clarity sake, only GP matrix multiplication is represented. It produces a partially encrypted circuit c_{GP} To obtain the encrypted circuit c', it necessary to use S and z too. The function to encrypt (1) is represented as a Boolean circuit c (2). The output matrix Y (3) is multiplied by GP (4). The result (5) is the partially encrypted output matrix Y_{GP} It can be represented by the corresponding Boolean equations (6) or as a "partially encrypted circuit" c_{GP} (7).

- Alice decrypts the result (algorithm D_A). She first removes permutation P: $y'P^{-1} = ySG + zP^{-1}$. Permuting individual error contributions does not change the Hamming weight of the vector and thus, $w(zP^{-1}) = w(z)$. The vector z is a correctable error vector since it is defined as: $w(z) = t$ exactly. The decoding algorithm for the code generated by G can correct an error with a weight of at most t, thus Alice is able to retrieve the cleartext result $y = c(x) \in Y$ and the error vector z from zP^{-1}.

- Alice finally performs the integrity verification (algorithm V_A): if $w(z) = t$, the output is accepted, tampering with the evaluation of c' is assumed otherwise. For integrity verification, the error weight is fixed. The maximum error weight that can be corrected using Goppa codes is t.

2.3 Scheme evaluation

Confidentiality of Execution

Confidentiality of execution relies on the hardness of retrieving the equations of the circuit c after their multiplication with matrix SGP and after adding the error z. First of all, an enumeration attack to recover the circuit c directly from c' is unfeasible using the code size proposed by McEliece ([n=1024, k=524, t=50]). Moreover this attacker requires the public key that is not even available to him here.

Retrieving the error circuit z from circuit c' is another possible attack. It is equivalent to trying to retrieving a subspace from a set of codewords with errors. In another context, this problem was termed Decision Rank Reduction [38] and was proven to be NP-complete. In order to avoid this attack our solution is based on errors with a Hamming weight equal to the maximum correction capability of the code.

Nonetheless, transformation E_A does not hide everything about circuit c. Bob can identify inputs (x_i, x_j) that have the same cleartext output $y_i = y_j$ because the distance $d(y_i', y_j')$ between the ciphertext values will be small so that errors remain correctable. In that case, $y_i' + y_j' = y_i \cdot SGP + z_i + y_j \cdot SGP + z_j = z_i + z_j \ (mod\ 2)$. An attacker would be able to recognize such values because the Hamming weight of their sum is at most equal to $2t$ even though he does not know the result cleartext value y_i. Differential cryptanalyse exploiting the fact that the error circuit z does not completely hide the linearity of the transformation were described in [13] and [7]. These attacks only apply when the public key is available but this one is kept secret in our scheme. Moreover, the identification of ciphertexts corresponding to the same cleartext can be suppressed [35] but implies an increase in the computational complexity of the encryption, decryption, and verification algorithms.

The majority voting attack described in [30], [39] exploits the non-deterministic nature of the cryptosystem to recover the secret code. The probability of success of this attack depends on getting a high number of different ciphertexts for each plaintext. This is not possible in our scheme that keeps secret the public key.

Integrity of execution

Let us establish the probability that the verifier accept an invalid y'. The set of acceptable cleartext outputs is $Y \subseteq \{0, 1\}^k$. Due to the definition of the error function, it is assumed that it is hard to establish any link between the inputs to the error function and the error patterns. Thus, picking a random encrypted output value $y' \in \{0, 1\}^n$ has a probability δ of being accepted. A's result y can only be valid if its value is an element of $Y \subseteq \{0, 1\}^k$. The probability of a successful attack can be calculated as "the probability of choosing an y existing in Y" times "the probability of generating a correct error weight". Generating a correct error means finding a t bit vector chosen at random among n bits. The worst case ($|Y| = 2^k$) leads to a probability of a successful attack of: $\delta \le 2^{k-n} \cdot \binom{n}{t}$. For a Goppa code [n=1024, k=524, t=101], the probability of a successful attack is: $\delta \le 2^{-215}$

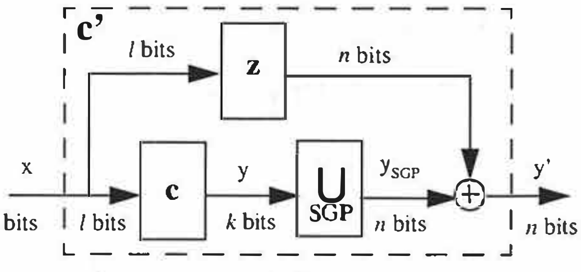


Figure 4: Modular implementation of the encrypted function

Circuit Size Evaluation

The circuit *c'* being evaluated by a remote host, it is important to minimize the impact of encryption on the circuit size, measured by its number of logical gates. Expansion rate cannot be calculated because it is specific to the structure of the original circuit. However, it is possible to study the worst case. The encrypted function can be implemented by a modular circuit *c'* as shown in Figure 4.

In practice, the encrypted function is implemented by a circuit based on simplified equations, the size of this circuit being necessarily smaller than that of the equivalent modular circuit shown in Figure 4. The circuit based on simplified equations offers the same functionality as the three different modules of Figure 4. Integrity and confidentiality properties of the protection scheme do not allow an attacker to retrieve the modular circuit from the equations. The size of the actual encrypted circuit is smaller than the sum of the sizes of the three modules:

$$Size_{c'} \le \underbrace{Size_{SGP} + Size_z + Size_{XOR}}_{\text{encryption represented as circuits } (\alpha)} + \underbrace{Size_c}_{\text{original circuit}}$$

The matrix *SGP* multiplication transforms the k cleartext circuit outputs y_i into n encoded outputs that are noted $y_{SGP,j}$: for instance, $y_{SGP,0} = 1 \cdot y_1 + (0 \cdot y_2) + 1 \cdot y_3 + (0 \cdot y_4) + 1 \cdot y_5$, and so on. For a given fixed number of inputs and outputs, the size of the *SGP*-encoding circuit ($Size_{SGP}$) is fixed (proportional to $k \cdot n$) and the size of the error circuit ($Size_z$) can be chosen. Both are independent of the original circuit size ($Size_c$) and thus $Size_{c'} \le Size_c + \alpha$. When the circuit is simple (e.g. $y = NOT(x)$ or $y=2x$), then $Size_c << \alpha$ and encrypting a circuit increases significantly the circuit size. However, when the circuit is large compared with the sizes of equivalent circuits for *SGP* multiplication and error, then $Size_c > \alpha$ and encrypting the circuit has a negligible effect on the circuit size increase: $Size_{c'} \approx Size_c$ (the size of the *XOR* function being negligible).

3 Protecting Functions within a Program

As explained in the previous section, function protection only allows the execution of one function while a program has to perform several functions in sequence. A naïve approach is to represent a program as a single Boolean circuit and protect this one using the function protection scheme described above. Unfortunately, this approach, which requires huge circuits, is totally unrealistic.

We propose a solution to the problem of software protection that consists in delegating the verification and decryption tasks (originally performed by the circuit owner *A*) to a trusted tamper-proof hardware (TPH) located at the untrusted site (a preliminary proposal was described in [26]). This TPH must be directly accessible by the untrusted host in order to suppress all interactions with the code owner (*A*) when the software is executed. The suggested solution assures the security of the circuits executed on an untrusted runtime environment. It also makes it possible to securely perform multi-step evaluation of functions. Additionally, this scheme ena-

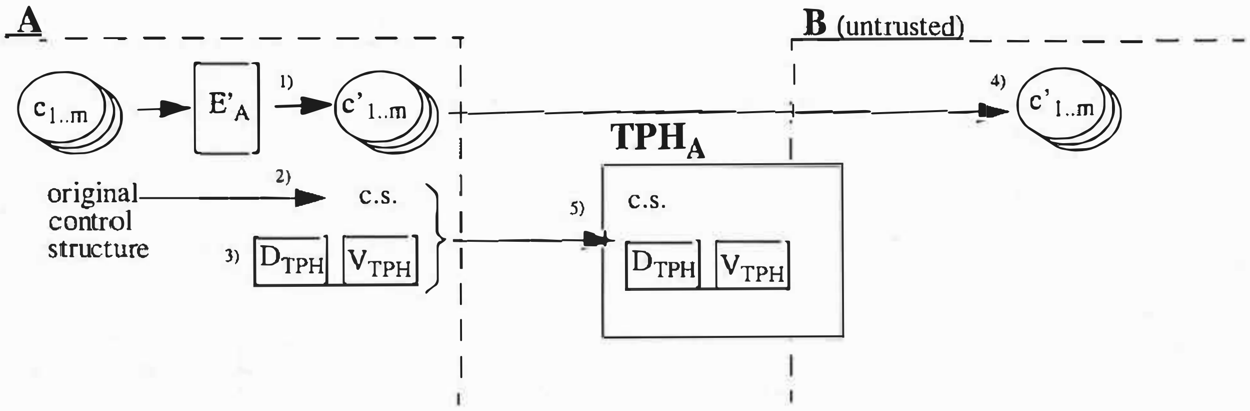


Figure 5: Installation of the program on the untrusted host and trusted TPH.
A has provided a *TPH* to *B*. 1) the functions used by the program (i.e. Boolean circuits in our implementation) are encrypted; 2) the control structure is modified to call those new functions; 3) the verification and decryption algorithm corresponding to the encryption are generated 4) the encrypted functions are sent to the untrusted host; 5) The control structure and the verification and decryption algorithms are sent to the *TPH* using a secure channel.

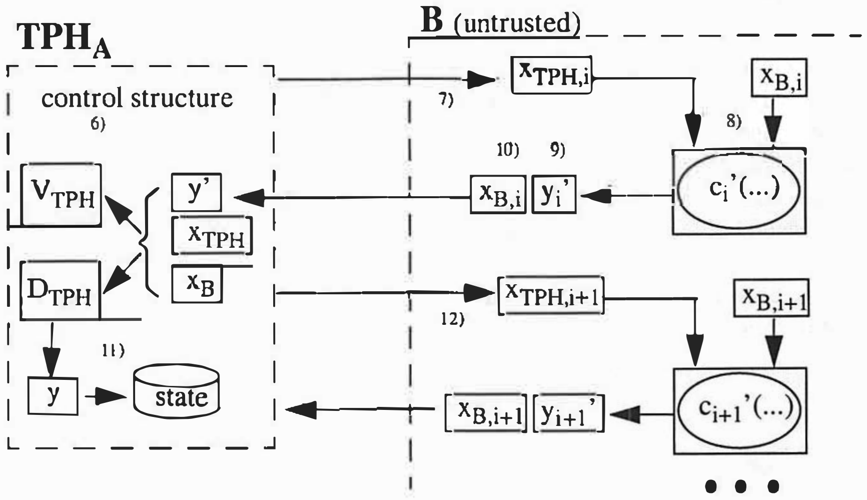


Figure 6: Evaluating an encrypted program on the untrusted host.
6) The *TPH* calls a function according to the control structure; 7) the TPH can input intermediate results; 8) the selected encrypted function is executed with data provided by B and/or TPH ; 9) The encrypted result is returned to the TPH; 10) the inputs are used to allow a more efficient verification (see section 3.2); 11) the temporary result is stored; 12) the next encrypted function is called ...

bles a program to deliver a cleartext result to the untrusted host without having to contact the code owner.

Assuming a wide deployment of mobile codes makes it unlikely that expensive tamper-proof hardware be used: this implies that the TPH will be limited in terms of storage and computational capacity. Even though our solution for multi-step execution is based on the protection technique described in the Section 2, it has to be adapted to cope with the computation power limitations imposed by the TPH (look at Section 3.2). The use of a TPH has already been suggested for delegating the functionality of a trusted party in specific contexts as can be seen in host-assisted secret key [8] or public key [17] cryptography applications. [6] proposes to separate a program into several pieces but does not deal with encrypted functions.

3.1 Computational Model

A program can be modeled as a set of functions plus a control structure, which defines the sequencing of functions. As in the previous section, functions are implemented with circuits. The computation of each individual circuit c_i depends on a set of inputs x received from the host and from the memory of the TPH. As before, the protection of each circuit from the untrusted environment where it is evaluated is achieved through its encryption. The control structure is uploaded to the tamper-proof hardware to protect it. Based on this control structure, the TPH instructs the untrusted environment to execute one of the encrypted circuits c_i'. For each output of circuit c_i', the TPH is able to verify the integrity of the result and to retrieve the cleartext result y in an efficient way. Each circuit c_i is encoded with a new algorithm E'_A.

A state of the computation can be maintained in the trusted TPH, in other words memory attacks need not be taken in consideration. It is mandatory that B receives a TPH trusted by A, which is not a very restrictive hypothesis. For instance, A could be a bank or an operator that provides a smart card to its client B, just like they already provide credit cards and SIM cards. A verification and decryption algorithm must be installed on the TPH via a secure channel, either before the TPH is distributed to clients or transmitted in encrypted form using a secret shared by A and her TPH.

Once the encrypted circuits (c'_1 ... c'_m) are installed (Figure 5) on host B, the TPH is in charge of choosing which function has to be evaluated and of providing a part of the inputs (x_{TPH}), the other part being provided by B (x_B). After each step (i.e. each encrypted function evaluation), the TPH deciphers and verifies the returned result (Figure 6). Note that a given function can be evaluated more than once with different inputs. When the TPH chooses the next encrypted function to execute (c_{i+n}), it provides input data ($x_{TPH,\ i+n}$). Those data are stored on the TPH.

3.2 Protection Scheme

The algorithms devised in the protection scheme of Section 2 have to be adapted to the new scenario in which the TPH has less computational power than the party that it represents. The algorithm used for function encryption remains the same since this operation is performed by the code owner, but the error computation is modified in order to simplify the verification performed by the TPH. The new verification and decryption algorithms are respectively called V_{TPH} and D_{TPH}. In order to simplify the verification and decryption, x_B is transmitted to the TPH.

The integrity of execution relies on the difficulty of creating forged pairs (x, y') that pass the verification process (x_{TPH} being known). Using the same terminology as in the previous section, this probability can be defined as follows:

if $y \neq c(x)$ then $P(V_{TPH}(x, y') = Accept) < \delta$
x being $(x_{TPH} \mid x_B)$

Error Circuit

As in the classical McEliece scheme, the function protection scheme of Section 2 introduced at most (and in our case, exactly) t errors into the encoded circuit: this represents the maximum number of correctable errors using the capacity of the code, since in the scheme, A did not know B's input x. This value is now retrieved on the TPH that also possesses the error circuit $z(x)$ and can entirely suppress the error without restraining to a specific correction capacity. It is thus possible to introduce much more weighted errors into the encrypted circuit.

The security parameter q, with $0 < q \leq n$, indicates the maximum weight of the error introduced. Using a Goppa code [n=1024, k=524, t=101], this parameter might be as high as 1024 bits, meaning that all bits of a given output y' might be in error, instead of t=101 bits as in the scheme of Section 2. In the general case, the number of errors introduced will be smaller than this upper bound, yet higher than the correctable case. This considerably limits enumeration attacks for retrieving the error circuit.

Nonetheless, the error circuit size must remain reasonable to retain any advantage from executing c' on the untrusted host rather than c on the TPH. For the construction of the error circuit, a trade-off should be found between the highest possible number n of simple error equations and a smaller number of more complex equations closer to a random error.

Result Decryption

In the new scheme, the decipherment is based on the inputs and outputs of the encrypted function evaluation. For each evaluation of circuit c', the TPH, which knows x_{TPH}, receives y' and x_B. The encrypted result can be written: $y' = ySGP + z(x_{TPH}|x_B)$. S, G, and P being known to the TPH, as well as the error circuit $z(x)$, it is possible to first compute and remove the error pattern:

$$ySGP = y' + z(x_{TPH}|x_B)$$

Since matrix G is in systematic form $(I \mid A)$, the GP encoding can be removed as follows:

$$ySG = yS(I|A) = (y' + z(x_{TPH}|x_B))P^{-1} \qquad (eq\ 1)$$

$$yS = ySI = \lceil (y' + z(x_{TPH}|x_B))P^{-1} \rceil_k \qquad (eq\ 2)$$

where $\lceil v \rceil_i$ is the vector formed by the first i bits of vector v.

The cleartext output y can finally be retrieved as follows:

$$y = \lceil (y' + z(x_{TPH}|x_B))P^{-1} \rceil_k \cdot S^{-1}$$

Integrity Verification

The verification algorithm is adapted to the new construction of the error circuit. Integrity of execution is ensured by controlling that all bits of the cleartext output are correct after having removed the supposed error pattern. Since the cleartext output y can be obtained by using only the first k bits of ySG, the remaining and redundant n-k bits are used to verify that the output computed by the untrusted host has not been tampered with.

From decryption *equation 1* (above), and using only the last n-k bits of ySG:

$$ySA = \lfloor (y' + z(x_{TPH}|x_B))P^{-1} \rfloor_{(n-k)}$$

where $\lfloor v \rfloor_i$ is the vector formed by the last i bits of vector v.

Since $ySA = (yS)A$, it can be deduced from decryption *equation 2* that the TPH needs only verify that the following equation is satisfied:

$$\left\lfloor (y' + z(x_{TPH}|x_B))P^{-1} \right\rfloor_{(n-k)} = \left\lceil (y' + z(x_{TPH}|x_B))P^{-1} \right\rceil_k \cdot A$$

3.3 Scheme Evaluation

Confidentiality of Execution

Evaluating the confidentiality relies on the same principles as in the previous section. Since it is now possible to introduce more weighted errors, the complexity of retrieving the initial circuit is increased in a ratio depending upon the chosen security parameter q.

A new problem is introduced by the multi-step execution concerning intermediate cleartext results. It sometimes happens that $x_{TPH,\,i} = D_{TPH}(y_{i-n}')$, that is, B can observe the cleartext result of a previously computed function. The fact that the cleartext result may be given back to the untrusted environment is critical. With a sufficient number of pairs of cleartext inputs and outputs, the untrusted host would be able to interpolate circuit c. For more details, refer to the limitations section below.

Integrity of Execution

An enumeration attack amounts to obtaining a forged pair (x, y') acceptable for the verifier. Since the TPH can get access to the input in addition to the encrypted output of circuit evaluation, the verification is now performed using the actual error pattern and not the error weight as before. The probability of such an attack being successful is thus even smaller than in the scheme of Section 2. Like for confidentiality, the use of more weighted errors can even further increase the difficulty of breaking the integrity of execution.

Moreover, part of the input is provided by the TPH and cannot be modified by the untrusted host: this makes it possible to obtain the equivalent of a variable error pattern for the inputs restricted to x_B, the untrusted host inputs. In other words, even if the untrusted host B is able to determine the error pattern for a given x_{TPH}, this pattern will not be useful for another value of x_{TPH}. In practice, x_{TPH} will vary for nearly each computation of a given function throughout the lifetime of a program.

Limitations

As to the limitations of this approach, it is obvious that the algorithmic structure of the protected program is not hidden: the repeated execution of a function can be traced. Private Information Retrieval techniques (PIR) [15] and oblivious RAM models [20] that hide the sequence of accesses provide a sophisticated solution to this problem. Unfortunately, those works have shown that hiding access patterns is prohibitively expensive. This is the reason why our scheme addresses the protection of each function used by a program rather than the protection of its algorithmic structure.

The result of some encrypted function f_i can be used as the input to another function f_j. This means that a malicious host can sometimes observe the cleartext result of one of the encrypted functions that form the building blocks of a program. Depending on the function and on the number of times it is evaluated, it might be possible to obtain enough cleartext inputs and outputs to interpolate the function. In order to avoid this problem a scheme using enciphered inputs and outputs could be used but the performance penalty is important.

However, even if the confidentiality of execution is partly broken, the integrity is not attacked. Indeed, interpolating a function allows an attacker to compute all corresponding cleartext outputs y, but knowing y and y' is already not sufficient to break the McEliece cryptosystem.

Implementability

Any function with fixed-length inputs and outputs can be represented as a Boolean circuit and thus encrypted by our scheme. However, to have an efficient implementation, it is necessary to define a computation model fitting the requirement of this approach. The TPH and the untrusted host have to support two different computing models:

- An algorithmic logic similar to what can be found in smart cards.
- A functional model supporting a representation of Boolean circuits. It could be implemented as truth tables (memory) or as circuits (programmable logic).

4 Related Work

This section presents other work related to the confidentiality or the integrity of execution. We also compare our solution with two other approaches, truth table encryption and gate-level encryption, that can be used to protect functions represented as Boolean circuits.

4.1 Confidentiality of Execution

Secure function evaluation is an instance of the more general problem of confidentiality of execution. Secure function evaluation has been addressed by many researchers ([41], [42], [19], [3], and [2], just to mention a few). Non-interactivity is an important requirement for mobile code, but the protocols addressing the circuit model need a round complexity dependent on the number of gates or depth of the circuit and are thus not well adapted to mobile code.

Sander and Tschudin [33], [32] defined what they called a function hiding scheme and focused on non-interactive protocols. In their framework, the privacy of *f* is assured by a encrypting transformation. The authors illustrated the concept with a method that allows computing with encrypted polynomials, based on the Goldwasser-Micali encryption scheme [18]. Sander and Tschudin took advantage of the homomorphic properties of the above encryption scheme to encrypt the coefficients of the polynomial, thus their technique does not hide the skeleton of the polynomial. Moreover, polynomials are not as expressive as Boolean circuits.

[36] presented a non-interactive solution for secure evaluation of circuits but which is restricted to log depth circuits (or NC^1 circuits). Protocols were designed for processing NOT and OR gates in a private way. The restriction on the depth of the circuit comes from the increase of the output size by a constant factor when computing an OR gate.

In [14] and [1], more efficient techniques are presented, that combine encrypted circuits [42] and one round oblivious transfers. However, the circuit expansion is high with this technique and this expansion compromises the narrow advantage in performance of mobile code as shown in [22].

4.2 Integrity of Execution

Integrity of execution is the possibility for the circuit owner to verify the correctness of the execution of his circuit. This problem has been extensively studied for achieving reliability (see for example [12] for a survey) but security requirements taking into account possible malicious behavior from the execution environment were not considered.

Other solutions cope with the maliciousness of the execution environment. Yee [44] suggested the use of proof based techniques, in which the untrusted host has to forward a proof of the correctness of the execution together with the result. Complexity theory shows how to build proofs for NP-languages and recently how to build Probabilistic Checkable Proofs (PCP) [4], [5]. PCP proofs require checking only a subset of the proof in order to assure the correctness of a statement. However, this subset has to be randomly determined by the checker, so the problem of using PCP proofs in our non-interactive scenario is that the prover has to commit to the overall PCP proof. We refer the interested reader to [21] for a comprehensive survey of the work on proofs.

In [11], the authors presented an interesting model for mobile computing and a solution that overcomes the problem of using PCP proofs. The agent is modeled as a probabilistic Turing machine, and the set of all possible states of this machine constitutes a NP language. There exists a verification process for language membership, that is, it is possible to check if an obtained state belongs to the language. This scheme relies on the use of non-interactive Private Information Retrieval techniques to avoid the transmission of the overall PCP proof of the specified language, the randomly chosen queries from the checker being encrypted. Our second scheme allows us to "trace" in real-time an execution step-by-step, one step being a function evaluation, and ensuring that each step is performed in accordance with the program semantics. In our scheme, verifying an execution does not require verifying a complex trace.

4.3 Encrypted Boolean Circuit Approaches

Boolean circuits can be protected from confidentiality and integrity attacks using three different encryption techniques: circuit encryption (our approach from Section 2), truth table encryption, and gate-level encryption [36], [19].

Circuit Encryption vs. Truth Table Encryption

Evaluating a function with a truth table simply corresponds to choosing the right line of the table that corresponds to the inputs and contains the outputs of a circuit. A simple protection of this scheme is to encrypt line-by-line each output of the truth table with a standard encryption algorithm. A new truth table with encrypted outputs is then obtained. In this approach, each result is pre-calculated.

The truth table outputs being encrypted line by line, the encrypted function it represents is by definition constructed randomly. Shannon showed that the size of almost every function with l inputs and one output is bigger than $2^l/l$. The size of the circuit implementing an encrypted truth table with n outputs can thus be assumed to be bigger than $n \cdot 2^l/l$ gates. This size does not depend on the initial cleartext circuit but essentially on the number of inputs and can be bounded: any function with l inputs and one output can be computed by a circuit of size $O(l \cdot 2^l)$ [40].

Our scheme modifies Boolean equations rather than outputs and, as shown in Section 2.3, $Size_{c'} = Size_c + \alpha$. It is possible to have $Size_{c'} \approx Size_c$ under reasonable assumptions about the size of c compared with the size of the error circuit or the SGP multiplication circuit (in a modular implementation). In the worst case, when the cleartext circuit size $Size_c$ is close to $l \cdot 2^l$, the size $Size_{c'}$ of the resulting encrypted circuit is not better than the size of an equivalent encrypted truth table.

Circuit Encryption vs. Gate Level Encryption

The gate-level encryption [36], [19] is a Computing with Encrypted Data scheme. Each gate of the circuit is replaced by a cryptographic module that use keys as inputs and outputs to represent *true* or *false* Boolean values. A function evaluation corresponds to cryptographic operations performed gate by gate. However, it is possible to observe the resulting construction and to deduce the initial circuit. This solution thus does not ensure confidentiality but only integrity. Valiant's universal circuit [37] makes it possible to see circuits as data. Thanks to it, it is possible to convert the Computing with Encrypted Data scheme into Computing with Encrypted Functions.

The main advantage of the gate-level encryption scheme is the linear impact of the encryption on the circuit size. Indeed, each gate of the initial circuit is replaced by one module and associated keys. When the universal circuit is used, the resulting size is $O(d \cdot s \cdot log(s))$ modules, where s is the size of the initial circuit and d its depth. This size increase is small and only depends on the initial circuit.

This approach has drawbacks: it is necessary either to interact with the circuit owner for each gate evaluation or to use oblivious transfers to provide inputs. Moreover, the scheme allows only one evaluation of the circuit otherwise the integrity, and confidentiality in case of use of a universal circuit, cannot be ensured.

In comparison, our approach is similar to the truth table in that it is an encryption of the whole output that is performed instead of a bit-by-bit encryption of the output. This makes it possible to evaluate a function more than once. Bit-by-bit encryption yields too much information about the circuit structure to permit two consecutive evaluations: a new encrypted circuit has to be recomputed after each evaluation.

5 Conclusion

This paper presented basic building blocks for securing mobile code executed in a potentially hostile environment. It first described a scheme that can autonomously evaluate a Boolean circuit in a potentially malicious environment. This scheme ensures at the same time the integrity and confidentiality of evaluation of the circuit. The protection is derived from the McEliece data encryption scheme, thus allowing an efficient encryption, verification and decryption. The original circuit is encrypted into a new circuit, which can be executed by an untrusted environment although its result can only be decrypted by the circuit owner. This scheme can generate an encrypted circuit with a size close to that of the original circuit. All functions implementable with Boolean circuits can be protected using this scheme.

Any program can be implemented as a single circuit and thus be protected using this function protection scheme. In practice however, that approach is totally unrealistic because of the huge size of the circuit. The second part of this article introduces another protection scheme that deals with programs rather than functions. This scheme resorts to using a tamper-proof hardware, albeit with a limited capacity compared with the program processing needs. The tamper-proof hardware protects the scheduling of a set of encrypted functions executed directly in the untrusted environment. The tamper-proof hardware also performs the result decryption and verification, which were previously done by the code owner.

The practical deployment of the latter scheme may finally be questioned because of the lingering requirement for a tamper-proof hardware. However, few years ago, authentication had similar needs and now such hardware is in wide use for authentication purposes. We envision the use of cheap tamper-resistant hardware like slightly modified smart cards as a possible solution.

References

[1] J. Algesheimer, C. Cachin, J. Camenisch, and G. Karjoth. Cryptographic security for mobile code. In *Proc. of the IEEE Symposium on Security and Privacy*, May 2001.

[2] Martín Abadi and Joan Feigenbaum. Secure circuit evaluation. *Journal of Cryptology*, 2(1):1–12, 1990.

[3] Martín Abadi, Joan Feigenbaum, and Joe Kilian. On hiding information from an oracle. *Journal of Computer and System Sciences*, 39(1):21–50, August 1989.

[4] Sanjeev Arora, Carsten Lund, Rajeev Motwani, Madhu Sudan, and Mario Szegedy. Proof verification and hardness of approximation problems. In *Proc. 33rd IEEE Foundations of Computer Science*, pages 14–23, October 1991.

[5] Sanjeev Arora and Shmuel Safra. Probabilistic checking of proofs: A new characterization of NP. *Journal of the ACM*, 45(1):70–122, 1998.

[6] D. Aucsmith. Tamper resistant software: an implementation. In *Proc. International Workshop on Information Hiding*, 1996. Cambridge, UK.

[7] Thomas A. Berson. Failure of the McEliece public-key cryptosystem under message-resend and related-message attack. In Burton S. Kaliski Jr., editor, *Advances in Cryptology—CRYPTO '97*, volume 1294 of *Lecture Notes in Computer Science*, pages 213–220. Springer-Verlag, 17–21 August 1997.

[8] Matt Blaze, Joan Feigenbaum, and Moni Naor. A formal treatment of remotely keyed encryption. In Kaisa Nyberg, editor, *Advances in Crytology - EUROCRYPT '98*, Lecture Notes in Computer Science, pages 251–265, Finland, 1998. Springer-Verlag.

[9] Manuel Blum and Sampath Kannan. Designing programs that check their work. In *Proceedings of the Twenty First Annual ACM Symposium on Theory of Computing*, pages 86–97, Seattle, Washington, 15–17 May 1989.

[10] Matt Blaze. High-bandwidth encryption with low-bandwidth smartcards. In Dieter Grollman, editor, *Fast Software Encryption: Third International Workshop*, volume 1039 of *Lecture Notes in Computer Science*, pages 33–40, Cambridge, UK, 21–23 February 1996. Springer-Verlag.

[11] Ingrid Biehl, Bernd Meyer, and Susanne Wetzel. Ensuring the integrity of agent-based computations by short proofs. In Kurt Rothermel and Fritz Hohl, editors, *Proc. of the Second International Workshop, Mobile Agents 98*, pages 183–194, 1998. Springer-Verlag Lecture Notes in Computer Science No. 1477.

[12] Manuel Blum and Hal Wasserman. Software reliability via run-time result-checking. *Journal of the ACM*, 44(6):826–849, November 1997.

[13] Anne Canteaut. *Attaques de Cryptosystèmes à Mots de Poids Faible et Construction de Fonctions t-Résilientes*. PhD thesis, Université Paris VI, October 1996.

[14] C. Cachin, J. Camenisch, J. Kilian, and Joy Muller. One-round secure computation and secure autonomous mobile agents. In *Proceedings of the 27th International Colloquium on Automata, Languages and Programming-ICALP 2000*, Geneva, July 2000.

[15] B. Chor, O. Goldreich, E. Kushilevitz and M. Sudan, Private information retrieval, *Proceedings of 36th IEEE Conference on the Foundations of Computer Science (FOCS)*, p. 41–50, 1995.

[16] C. Collberg, C. Thomborson and, D. Low, A taxonomy of obfuscating transformations, Technical Report 148, Department of Computer Science, University of Auckland, 1996.

[17] Joan Feigenbaum. Locally random reductions in interactive complexity theory. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 13:73–98, 1993.

[18] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, April 1984.

[19] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, pages 218–229, New York City, 25–27 May 1987.

[20] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, May 1996.

[21] Oded Goldreich. *Modern Cryptography, Probabilistic Proofs and Pseudorandomness*. Springer-Verlag, 1999.

[22] Daniel Hagimont and Leila Ismail. A performance evaluation of the mobile agent paradigm. In *Proceedings of the Conference on Object-Oriented*

Programming, Systems, Languages and Applications, pages 306–313, Denver-USA, November 1999.

[23] Sergio Loureiro and Refik Molva. Function hiding based on error correcting codes. In Manuel Blum and C. H. Lee, editors, *Proceedings of Cryptec'99 - International Workshop on Cryptographic Techniques and Electronic Commerce*, pages 92–98. City University of Hong-Kong, July 1999.

[24] Sergio Loureiro and Refik Molva. Privacy for Mobile Code. In *Proceedings of the Distributed Object Security Workshop* - OOPSLA'99, pages 37-42, Denver, November 1999.

[25] Sergio Loureiro, Refik Molva, and Yves Roudier. Mobile code security. *In Proceedings of ISYPAR'2000, 4ème Ecole d'Informatique des Systèmes Parallèles et Répartis*, Toulouse, France, February 2000.

[26] Sergio Loureiro and Refik Molva. *Mobile Code Protection with Smartcards*. In 6th ECOOP Workshop on Mobile Object System. Cannes. France. June 2000.

[27] R. J. McEliece. The theory of information and coding, *Encyclopedia of Mathematics and its Applications*, Vol. 3, Addison-Wesley, Reading, MA, 1977.

[28] R. McEliece. A public-key cryptosystem based on algebraic coding theory. In *Jet Propulsion Lab. DSN Progress Report*, 1978.

[29] Silvio Micali. CS Proofs (extended abstract). In *IEEE Proceedings of Foundations on Computer Science*, pages 436--453, 1994.

[30] Joost Meijers and Johan van Tilburg. Extended majority voting and private-key algebraic-code encryptions. In Hideki Imai, Ronald L. Rivest, and Tsutomu Matsumoto, editors, *Advances in Cryptology—ASIACRYPT '91*, volume 739 of *Lecture Notes in Computer Science*, pages 288–298, Fujiyoshida, Japan, 11–14 November 1991. Springer-Verlag. Published 1993.

[31] Nicolas Sendrier. Efficient generation of binary words of given weight. In Colin Boyd, editor, *Cryptography and Coding; proceedings of the 5th IMA conference*, number 1025 in Lecture Notes in Computer Science, pages 184–187. Springer-Verlag, 1995.

[32] Tomas Sander and Christian Tschudin. On software protection via function hiding. In *Proceedings of the Second Workshop on Information Hiding*, Portland, Oregon, USA, April 1998.

[33] Tomas Sander and Christian Tschudin. Towards mobile cryptography. In *Proceeding of the 1998 IEEE Symposium on Security and Privacy*, Oakland, California, May 1998.

[34] Luis F. G. Sarmenta. *Volunteer Computing*. Ph.D. thesis. Dept. of Electrical Engineering and Computer Science, MIT, March 2001.

[35] Hung-Min Sun. Improving the security of the McEliece public-key cryptosystem. In *Proceedings of Asiacrypt 98*, pages 200--213, 1998.

[36] Tomas Sander, Adam Young, and Moti Yung. Non-interactive cryptocomputing for NC1. In *Proceedings of the IEEE FOCS*, October 1999.

[37] Leslie G. Valiant. Universal circuits (preliminary report). In *Conference Record of the Eighth Annual ACM Symposium on Theory of Computing*, pages 196-203, Hershey, Pennsylvania, 3-5 May 1976.

[38] A. Valembois. Recognition of binary linear codes as vector-subspaces. In *Workshop on Coding and Cryptography'99, Book of abstracts*, pages 43–51, Paris, France, January 1999.

[39] Johan van Tilburg. *Security-Analysis of a Class of Cryptosystems Based on Linear Error-Correcting Codes*. PhD thesis, Technische Universiteit Eindhoven, 1994.

[40] Ingo Wegener. The Complexity of Boolean Functions. Eiley-Teubner, 1987.

[41] A.C. Yao. Protocols for secure computations. In *IEEE Symposium on Foundations of Computer Science 82*, pages 160–164, Chicago, 1982.

[42] A.C. Yao. How to generate and exchange secrets. In *IEEE Symposium on Foundations of Computer Science 86*, pages 162–167, Toronto, 1986.

[43] Bennet Yee. Using Secure Coprocessors. Technical Report CMU-CS-94-149. School of Computer Science, Carnegie Mellon University. May 1994.

[44] Bennet Yee. A sanctuary for mobile agents. Technical Report CS97-537, UC at San Diego, Dept. of Computer Science and Engineering, April 1997.

MICROCAST: Smart Card Based (Micro)pay-per-view for Multicast Services

Josep Domingo-Ferrer, Antoni Martínez-Ballesté and Francesc Sebé
Universitat Rovira i Virgili
Dept. of Computer Engineering and Maths
Av. Països Catalans 26, E-43007 Tarragona, Catalonia
e-mail {jdomingo,anmartin,fsebe}@etse.urv.es

Abstract

With the increased availability of broadband fixed and mobile communications, multicast content delivery can be expected to become a very important market. Especially for wireless multicast delivery, it is important that payment collection be fine-grain: the customer should pay only for the content that she actually consumes. This can be achieved by using pay-per-view based on micropayments. This paper proposes the first method for enabling pay-as-you-watch services in a multicast content delivery environment. On the customer's side, micropayment generation is implemented in a smart card which can be plugged into the customer's receiving device (computer, digital video receiver, PDA, mobile phone, etc.). Micropayment collection and verification are distributed among multicast routers, which avoids bottlenecks inherent to many-to-one payment transmission.

Keywords: *Multicast delivery, Pay-per-view, Pay-as-you-watch, Micropayments, Smart cards in the Internet.*

1 Introduction

Communication technologies have been evolving in many important aspects over the last few years. On one hand, broadband communications such as city-wide WLANs, ADSL, cable networks and UMTS are becoming widespread. On the other hand, audio and video compression codecs such as DivX, Realmedia, etc. improve the use of the available bandwidth. Finally, the appearance of mobile phones with high-resolution color display and Internet-enabled PDA's will bring brand new multimedia services to everybody, everywhere. There are great opportunities to create a huge market for multimedia content delivery, featuring news broadcasting, videoconferencing, movie channels, on-line gambling, etc. Consequently, it seems natural to use mobile communications and portable devices, along with traditional desktop PC's, as new privileged outlets for digital content delivery in pay-per-view mode. Smart card based micropayments stand out as one of the most promising solutions to obtain a fine-grain fee collection service: the customer uses her smart card to perform micropayments as content is being received.

Most multimedia delivery services operate in multicast mode to send content over the Internet. By using multicast, one single data stream can reach hundreds, thousands and even millions of target media players.

1.1 Contribution and plan of this paper

This paper describes a method for enabling pay-per-view services in a multicast content delivery environment. On the customer's side, micropayment generation is implemented in a smartcard which can be plugged into the customer's receiving device (computer, digital video receiver, PDA, mobile phone, etc.). Micropayment collection and verification are distributed among multicast routers, which avoids bottlenecks inherent to many-to-one payment transmission.

Section 2 gives some background on multicast communication; the use of pay-per-view and micropayments in multicast is also approached. Section 3 describes the architecture of MICROCAST, a system for pay-per-view multicast content delivery. The MICROCAST micropayment protocol suite is fully described in Section 4. Finally, Section 5 contains some conclusions and suggestions for future work.

2 Multicast communication

Depending on the number of receivers, three types of communication can be distinguished:

- Unicast communication: one source and one receiver.
- Broadcast communication: one source node and *all* remaining nodes acting as receivers. As an example, consider video broadcast in a LAN: the same data are streamed from the source to the entire network by using the broadcast IP address.
- Multicast communication: one source and a group of receivers. As an example, consider a local digital cable TV network, where a particular piece of video content is to be distributed only to subscribers who are paying for it (rather than to the entire neighborhood).

2.1 Multicast group management

If a source is to communicate with n receivers, one could naively think of using n unicast communications (which results in the source being an output bottleneck) or one broadcast channel (which results in the entire network being flooded). Both solutions are wasteful in terms of bandwidth. It should be noted that the Internet is nowadays already full of millions of IP packets only controlled by their time-to-live or by the TCP protocol.

A better option to avoid increasing network congestion is for receivers to join a multicast group and have the content sent to them by using their multicast group IP address [5, 14]. A multicast group G is a set of receivers that are interested in receiving a particular kind of information.

Efficient multicast design and implementation is currently an open issue. The multicast task is carried out by multicast routers, which join previously established multicast groups identified by a multicast IP address[1]. These routers are capable of sending the data flow to multicast group G.

The basic tasks to be performed in multicast communication are: advertise the multicast session, manage group enrollment by the customers who want to receive the stream and, concurrently to group enrollment, build the multicast routing tree. Several multicast protocols have been proposed in the literature, such as MOSPF[6], PIM-DM[3], PIM-SM[2].

2.2 Pay-per-view and pay-as-you-watch

The name "pay-per-view" is certainly misleading. In current digital TV platforms, a fixed monthly fee is paid to subscribe to a basic package of channels and services. It is also possible to view some special "pay-per-view events" (*e.g.* movies, football matches) by paying ***in advance*** the price corresponding to the event. This form of pay-per-view means that the content is viewed ***after*** the customer has paid. There are at least two problems with the fee collection scheme just described. One problem is that the customer pays for a basic offer that is usually expensive for her. The other problem is that, in pay-per-view events, the customer pays for the whole piece of content: if she wants to stop watching anytime, she is losing a part of her money. Pay-per-view as contents are being streamed from the server to the customer (*pay-as-you-watch*) seems an option that fits better the customer needs. Successive payments can be performed every minute, for example. If a customer switches her player off, she only has paid for the minutes viewed so far. Of course, these frequent payment will be small ones, so credit card transactions or electronic payment systems like SET are too expensive, too complicated or both [7].

2.3 Micropayments

The operating costs of standard electronic payment systems are unaffordable for small amounts and can be split into communication and computation costs, the latter being caused by the use of complex cryptographic techniques such as digi-

[1]Multicast addresses are IP numbers in the range between 224.0.0.0 and 239.255.255.255

tal signatures. Micropayments are electronic payment methods specifically designed to keep operating costs very low. In most micropayment systems in the literature, computational costs are dramatically reduced by replacing digital signatures with hash functions[11]. For example, this is the case of PayWord and Micromint[9], where the security of coin minting rests on one-way hash functions.

The main barrier to using traditional micropayment schemes for fee collection in multicast environments is their lack of scalability: a large number of receiving subscribers eventually overload the source with payment implosion. The MICROCAST protocol achieves scalability by distributing the effort of micropayment collection and verification among multicast routers. Unlike traditional micropayment schemes, MICROCAST does not concentrate on minimizing computation for micropayment generation and verification. By requiring micropayments to be less frequent (say every few minutes) and verification to be distributed, MICROCAST can still use short-exponent discrete exponentiations and provide the content source with a proof that every customer has paid.

More specifically, the scalability of our system is based on the following properties not fulfilled by conventional micropayment schemes (which are inherently unicast):

Aggregation Payments collected by routers at one level of the multicast tree can be aggregated and forwarded to the next upper level towards the source. Each aggregation only requires one product and one addition.

Single-step verification Verifying an aggregated payment can be done in a single step. There is no need to verify each individual payment included in the aggregation, which would imply non-scalability. Payment verification requires one short-exponent exponentiation, but this is no problem, since verification is performed only once per micropayment period by each tree node (regardless of the number of its child nodes).

Note 1 As it can be seen in Section 4.4 below, using the discrete exponentiation as a one-way function is justified by its homomorphic properties, which allow payment aggregation and single-step verification and are not shared by the (faster) one-way hash functions.

2.4 Rekeying

Multicast routers form a group that receives a multicast data stream. The router will possibly send the info to a hub that floods all its output connections, thus making the information reach every node in the subLAN, including nodes whose customers have not paid for the content. Cryptography should be used to prevent cheaters from being able to view the content by using packet sniffers. Customers in the multicast group have a decoding key to be able to decode the content they receive.

Hence, legitimate customers are those who pay every multicast period t (a multicast period typically lasts a few minutes). When a customer does not pay, she will be considered non-legitimate; in this case, a rekeying procedure will start which consists of distributing a new decoding key to every remaining legitimate customer. As a result, the removal of a group member will involve as many unicast transmissions as legitimate customers remain in the group. Fortunately, rekeying reaches a maximum cost of $O(\log n)$ when using tree structure controls[12, 1]. Even if rekeying is an important multicast issue, the reader of this paper only needs to keep in mind that it is the procedure started when a customer in a multicast group is removed due to lack of valid payment or when a new customer joins the group.

3 MICROCAST architecture

As it was pointed out in Section 1, MICROCAST is a pay-as-you-watch system for multicast content delivery. A typical application for MICROCAST could the pay-as-you-watch video distribution to thousands of customers. By using her smart card plugged into her video receiver, a customer can join a multicast group when she is interested in watching an event. After joining a group, the customer makes a micropayment every period t to keep watching the event.

In a conventional micropayment system, a bottleneck would arise at the video source as a result of micropayment collection, because thousands of coins arrive every period[2]. RFC 3170[8] on multicast applications recommends that multicast proto-

[2]A coin can be a 200-bit vector, and period t is short

cols should be able to use the multicast router link to provide bidirectional communications instead of using unicast channels to communicate receivers with the source. The MICROCAST architecture follows that design principle: multicast routers handle customers and coins, which results in a dramatical reduction of the amount of payment data sent to the content source.

The MICROCAST system consists of a source, a set of multicast routers, the customer smart cards and receivers, a rekeying system and a bank (see Figure 1). Each component is described next:

- **Source.** The source is the provider of the multicast content. Typical sources can be a movie channel, a news service, a music station, a sports service, etc. The source sells the content to thousands, even millions, of potential customers. As content is being delivered, the source expects some kind of payment from customers or at least something that certifies whether each particular customer is currently paying.
- **Multicast router.** The router in the multicast tree also acts as a micropayment subcollector. It requests micropayment from its customers and, after a timeout, it collects and verifies customer micropayments. Then, the router forwards valid payments to his parent router in the multicast tree, in order for payment information to reach the source (or the main micropayment collector, depending on the business model).
- **Customer device.** The customer receiving device (say a digital video receiver) is smart card enabled. Firstly, the smart card certifies through some easy calculations (see Section 4) that a payment is done by the customer. Thus, the role of the smart card is twofold: 1) authenticate payment origin; 2) help enforcing subscription certificate revocation when the customer is not backed by enough money in her bank account.
- **Rekeying system.** The rekeying system maintains a structure of legitimate customers (those who pay) and generates and distributes a new decoding key whenever any router of the system informs that some customer has failed to pay.

enough to keep payment fine grain, *i.e.* for content reception and payment to progress nearly concurrently.

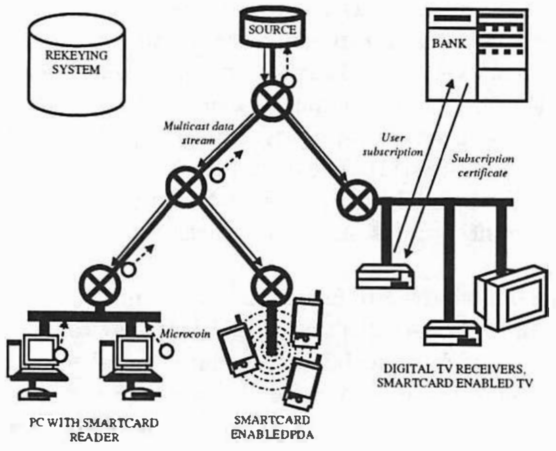


Figure 1: System architecture for MICROCAST

- **Bank.** The bank's role is to act as certification authority for customers. If a customer has enough funds in her account, the bank gives her a kind of public key (see Section 4).

4 The MICROCAST protocol suite

The MICROCAST protocol suite consists of protocols for bank setup, customer subscription, multicast session join, micropayment, customer removal, coin redemption and subscriber certificate revocation.

4.1 Bank Setup

As can be inferred from the previous section, the bank is a trusted party. It has a public/private key pair which is used to issue customer subscription certificates. The bank setup protocol works as follows:

Protocol 1 (Bank setup) *The bank does:*

1. *Choose a random prime q such that $2^{159} < q < 2^{160}$. All exponents in the remaining protocols will be modulo q, so we take a relatively small q, like it is done in the Digital Signature Standard (DSS,[15]).*

2. *Choose two large RSA primes p_1 and p_2, such that $2^{511} < p_1, p_2 < 2^{512}$ and q divides $p_1 - 1$. Compute an RSA modulus $n = p_1 p_2$ [10].*

3. *Randomly choose an RSA public exponent e and compute the corresponding private exponent d, so that $ed = 1 \bmod \phi(n)$*

4. *Compute a generator g of a cyclic subgroup of $\mathbb{Z}_n^*$ having order q. See the DSS key generation algorithm [15] on how to find a generator of a subgroup with a specified order.*

5. *Publish n, q, e and g in a publicly available directory.*

We next show that publication of q does not turn factoring n into an easy problem. After Protocol 1, an intruder knows q, which is a divisor of $p_1 - 1$. Equivalently, r exists such that $qr + 1 = p_1$. Note that the intruder does not know p_1. Therefore, the only strategy to find r is by brute search until an r is found such that $qr + 1$ is a divisor of n. Now, according to Protocol 1, $2^{159} < q < 2^{160}$ and $2^{511} < p_1 < 2^{512}$; therefore, the intruder only knows that $2^{351} < r < 2^{353}$, so brute search of r is computationally infeasible.

4.2 Customer subscription

In order to be able to use the system, a customer needs a subscription certificate. Through this certificate, the bank certifies that the customer has a bank account which backs the customer's payments. Subscription certificates are only valid for a period T (*e.g.* one day) and are generated using the protocol below. Short certificate validity periods allow implicit revocation to be used in the way explained in [4]. The duration of period T is a trade-off between the cost of key generation and the risk of revoked keys being re-used as described in [4].

Protocol 2 (Customer subscription)

1. *Customer U is assigned a unique system identifier Id_U.*

2. *Customer U's Smart Card SC_U holds a symmetric encryption key k_U and generates a batch of public/private key pairs which are certified using the asynchronous certification technique based on certificate verification trees (CVTs) described in [4]. Specifically, each key pair in the batch corresponds to a different certificate validity period T (*e.g.* *a key pair per day) and is computed as follows:*

   (a) *SC_U generates a random private key α_U^T such that $0 < \alpha_U^T < q$.*

   (b) *The corresponding public key is computed as $P_U^T = g^{\alpha_U^T} \bmod n$.*

   (c) *SC_U encrypts α_U^T using k_U and sends $(P_U^T, E_{k_U}(\alpha_U^T))$ to the bank.*

3. *For each public key received, the bank generates a certificate statement C_U^T containing the following data: customer identifier Id_U, public key P_U^T corresponding to period T, certificate expiration date (end of period T).*

4. *Following [4], all newly generated certificates in the batch are added to the publicly available certificate verification tree in the next CVT update. The root of the updated CVT is RSA signed with the private key d of the bank.*

The security of the private keys α_U^T generated in Protocol 2 is based on the difficulty of computing discrete logarithms in the subgroup of size q generated by g. This problem is similar to the modulo q discrete logarithm problem used in [15].

4.3 Multicast session join

Let us assume that, at micropayment period t and at public key validity period T, a set of customers wish to join a multicast session S (see Section 3 for an explanation of what a micropayment period is). To keep the discussion simple and without loss of generality, we assume that a session starts and ends within the same public key validity period T. The following joining protocol is used:

Protocol 3 (Session join)

1. *For each customer U in the set of new customers do:*

   (a) *U sends to the micropayment collector (typically the content source) her certificate C_U^T for the current public key validity* *period T. The certificate is obtained by U from the CVT and the corresponding encrypted private key is obtained from the bank ([4]). SC_U decrypts $E_{k_U}(\alpha_U^T)$ and obtains α_U^T.*

(b) *The micropayment collector verifies the validity of C_U^T.*

(c) *If C_U^T is valid and authentic, the micropayment collector responds with a message containing the following data: session identifier Id_S, session start date and time $(Date_S, Time_S)$, and value $Value_S$ of each micropayment.*

2. *The micropayment collector (or the content source) includes in the multicast distribution tree all new customers U whose certificates C_U^T have been successfully validated. The tree reflects the location of routers and subscribers (customers).*

3. *The micropayment collector sends to each router R in the tree the following information:*

   - *For every new subscriber U that is a child of R in the tree, the current valid subscriber's public key P_U^T.*
   - *For every router R_i that is a child of R in the tree, the aggregated key of all descendant subscribers of R_i in the tree, computed as*

   $$P_{R_i}^t := P_{R_i}^{t-1}\Big(\prod_{U\in desc(R_i,t)} P_U^T\Big) \bmod n$$

   where $desc(R_i,t)$ is the subset of new customers being placed under R_i during micropayment period t. (If the multicast session starts at micropayment period t, we take $P^{t-1} := 1$). Note that

   $$P_{R_i}^t = P_{R_i}^{t-1} g^{\sum_{U\in desc(R_i,t)} \alpha_U^T} \bmod n$$

4.4 Micropayment protocols

Every time a period t finishes, a customer must perform a micropayment to keep receiving the content during the next period. The micropayment collector (*i.e.* the router in charge of a group of customers) asks all customers in his group to perform the next payment as follows:

Protocol 4 (Micropayment request) *The micropayment collector does:*

1. *Compute $x_t := \mathcal{H}(Id_S, Date_S, Time_S, Value_S, t)$ where $\mathcal{H}(\cdot)$ is a one-way hash function.*
2. *Generate the micropayment request message for period t as (x_t, t) and multicast it to all customers in the group.*

Customers in a group react to micropayment request by generating a coin using the protocol below:

Protocol 5 (Coin generation) *The smart card of each customer U in a group does:*

1. *Upon receiving the micropayment request for period t, check*

   $$x_t \stackrel{?}{=} \mathcal{H}(Id_S, Date_S, Time_S, Value_S, t)$$

2. *Generate a random $a_U^t \in \{1,\cdots,q-1\}$ and compute $A_U^t := g^{a_U^t} \bmod n$*
3. *Compute $p_U^t := \alpha_U^T + x_t \cdot a_U^t \bmod q$*
4. *The coin for the micropayment corresponding to period t is the tuple $coin_U^t := (Id_U, A_U^t, p_U^t, x_t)$*
5. *Send $coin_U^t$ up to the parent router.*

Coins are generated by customers that correspond to the multicast tree leaves. In the last step of Protocol 5, coins are sent by customers to parent routers. Such routers check the validity of the received coins and aggregate valid coins. Aggregation uses the homomorphic property of the discrete exponentiation (namely that $g^x \cdot g^y = g^{x+y}$), which is one strong argument in favor of using the discrete exponentiation as one-way function (see Note 1). The aggregated coin is then forwarded by the verifying router to his parent router and so on up to the tree root (micropayment collector). Thus, depending on its level, a router can receive two kinds of coin:

- A single coin from a customer leaf node directly connected to the router.
- An aggregated coin from a direct child router (see Protocol 6 below for a description of how coins are aggregated).

The protocol to aggregate coins by an intermediate router is as follows:

Protocol 6 (Coin aggregation)

1. *Initialize the new aggregated coin as*

$$coin_R^t = (list_R^t, A_R^t, p_R^t, x_t) := (\{\}, 1, 0, x_t)$$

2. *For each single coin received from customer U_i, that is, $coin_{U_i}^t = (Id_{U_i}, A_{U_i}^t, p_{U_i}^t, x_t)$, do*

$$coin_R^t := (list_R^t \cup \{Id_{U_i}\},$$
$$(A_R^t \cdot A_{U_i}^t) \bmod n, (p_R^t + p_{U_i}^t) \bmod q, x_t)$$

3. *For each aggregated coin received from router R_i, that is, $coin_{R_i}^t = (list_{R_i}^t, A_{R_i}^t, p_{R_i}^t, x_t)$, do*

$$coin_R^t := (list_R^t \cup list_{R_i}^t,$$
$$(A_R^t \cdot A_{R_i}^t) \bmod n, (p_R^t + p_{R_i}^t) \bmod q, x_t)$$

4. *Send the aggregated coin $coin_R^t$ up to the parent router.*

The protocol to check coin validity is:

Protocol 7 (Coin validity check)

1. *If a single coin is received from customer U_i, the following check is performed:*

$$P_{U_i}^T \cdot (A_{U_i}^t)^{x_t} \overset{?}{\equiv} g^{p_{U_i}^t} \pmod n \qquad (1)$$

Note that Check (1) is consistent with the structure of coins constructed by Protocol 5.

2. *If an aggregated coin is received from a child router R_i, the following check is performed:*

$$P_{R_i}^t \cdot (A_{R_i}^t)^{x_t} \overset{?}{\equiv} g^{p_{R_i}^t} \pmod n \qquad (2)$$

Lemma 1 *Assuming that the discrete logarithm problem as sketched in Protocol 1 and the RSA problem are difficult, coin forgery by an intruder is infeasible.*

Proof: To impersonate customer U and mint a single coin belonging to U at public key validity period T, an intruder knows P_U^T, x_t, g and n and must find A_U^t and p_U^t such that Equation (1) is satisfied. There are two ways to proceed:

1. Follow Protocol 5. In that case, the intruder must know the legitimate customer's private key α_U^T (which is protected by the difficulty of the discrete logarithm problem over the subgroup generated by g).

2. Generate a random p_U^t and compute A_U^t as

$$A_U^t := (g^{p_U^t} P_U^{T^{-1}} \bmod n)^{x_t^{-1} \bmod \phi(n)} \bmod n$$

Now, given x_t, computing $x_t^{-1} \bmod \phi(n)$ without knowing the factorization of n is the RSA problem.

Forging an aggregate coin that satisfies Equation 2 is analogous. □

Lemma 2 *The security of a customer's private key does not decrease as the number of coins she mints increases.*

Proof: Without loss of generality, compare the situation where one coin has been minted with the situation where two coins have been minted. Assume one coin has been generated during micropayment period t_1 and public key validity period T. Then the following equation holds:

$$p_U^{t_1} \equiv \alpha_U^T + x_{t_1} a_U^{t_1} \pmod q$$

where, only $p_U^{t_1}$ and x_{t_1} are known to possible intruders (such quantities are part of the coin). Thus, there is one equation and two unknowns α_U^T and $a_U^{t_1}$, so α_U^T cannot be determined. If a second coin is generated during micropayment period

$$p_U^{t_2} \equiv \alpha_U^T + x_{t_2} a_U^{t_2} \pmod q$$

the number of equations increases to two, but there are now three unknowns α_U^T, $a_U^{t_1}$ and $a_U^{t_2}$. In general, it can be seen that generation of m coins results in m equations with $m+1$ unknowns, one of which is the customer's private key α_U^T. □.

4.5 Customer removal

Customer removal from a group is caused by lack of valid payment. There may be two situations behind the lack of valid payment: 1) a customer does not send any coin to her parent router; 2) a customer sends an invalid coin to her parent router. Both situations are handled by the following protocol:

Protocol 8 (Customer removal)

1. [Routers with child customers] *When a router R in the level previous to customers receives and forwards the micropayment request to its child customers, R starts a timer. Upon timer expiration, all child customers U of R not having sent valid coins are supposed to have left the group. Only valid coins will be aggregated by R and sent up to its parent router. Child customers U having failed to provide valid coins will have their public keys P_U^T removed from R's memory.*

2. [Routers with child routers] *Each intermediate route R in the path to the root of the multicast tree aggregates valid coins received from child routers R_i and forwards the aggregated coin to its parent router. In order to check the validity of coins using Equation (2), the intermediate router R needs to update the public key of each child router R_i as follows:*

   (a) *If $list^t_{R_i}$ in $coin^t_{R_i}$ is the same as $list^{t-1}_{R_i}$ in $coin^{t-1}_{R_i}$, then $P^t_{R_i} := P^{t-1}_{R_i}$.*

   (b) *If $list^t_{R_i} \neq list^{t-1}_{R_i}$, then obtain $P^t_{R_i} := P^{t-1}_{R_i} \cdot (P^T_{U_i})^{-1} \bmod n$ for all customers U_i who were in $list^{t-1}_{R_i}$ but are not in $list^t_{R_i}$. (Only removed customers are dealt with by this protocol, new customers being handled by Protocol 3).*

3. [Root node] *When the last aggregated coin reaches the root node (micropayment collector), the computations performed are the same described for intermediate nodes. In addition, the root node starts a multicast rekeying procedure if $list^t_{Root} \neq list^{t-1}_{Root}$,* i.e. *if customers need to be removed.*

4.6 Coin redemption

During a multicast session, the micropayment collector stores the final aggregated coin for each performed micropayment. This coin contains a list including the identifiers of all customers that performed a payment. When the number of collected coins is large enough, the micropayment collector contacts the bank in order to redeem them.

Protocol 9 (Coin redemption)

1. *When the aggregated coin corresponding to period t has to be redeemed, the micropayment collector sends to the bank the session identifier Id_S, the session start date and time $(Date_S, Time_S)$, the value $Value_S$ of individual coins in that session, the micropayment period t, and the final aggregated coin*

$$coin^t_{Root} = (list^t_{Root}, A^t_{Root}, p^t_{Root}, x_t)$$

2. *The bank does:*

   (a) *If this is the first coin redemption of the current multicast session then check that the public key of all customers in the list field is correctly certified and compute $P^t_{Root} := (\prod_{U_i \in list^t_{Root}} P^T_{U_i}) \bmod n$.*

   (b) *If this is not the first coin redemption of the session, then*

      i. *If $list^t_{Root}$ in $coin^t_{Root}$ is the same as $list^{t-1}_{Root}$ in $coin^{t-1}_{Root}$, then $P^t_{Root} := P^{t-1}_{Root}$.*

      ii. *If $list^t_{Root} \neq list^{t-1}_{Root}$, then obtain P^t_{Root} as the modulo n product of P^{t-1}_{Root} times the public keys of the new customers times the multiplicative inverses of the public keys of the removed customers.*

   (c) *Compute*

   $$x_t := \mathcal{H}(Id_S, Date_S, Time_S, Value_S, t)$$

   (d) *Check* $P^t_{Root} \cdot (A^t_{Root})^{x_t} \overset{?}{\equiv} g^{p^t_{Root}} \pmod n$

   (e) *If all checks are correct, transfer the appropriate amounts from each customer account to the account of the micropayment collector (or directly to the account of the multicast content source, depending on the business model). In order to avoid performing microtransfers from each customer, a better strategy is to cluster several successive micropayments and perform a larger transfer from each customer.*

4.7 Subscription certificate revocation

It is possible for a customer to run out of funds before all of her certificates expire. In this situation, it would be possible for her to perform micropayments not backed by enough funds in her bank account. This situation is detected by the bank during coin redemption. In this case, the bank would revoke all her subscription certificates for future time intervals. The implicit revocation mechanism described in [4] is used: the bank does not supply any more encrypted private keys to the customer's smart card for joining the session in subsequent periods (Protocol 3).

5 Conclusions and future work

A micropayment protocol suite for multicast pay-per-view content delivery has been presented. The customer is represented by her smart card in all protocols in the suite. The proposed scheme has been simulated and works well as long as synchronization between customers is maintained. The main goal of the protocol is to to distribute micropayment collection so as eliminate the bottleneck associated to Mto1 applications. Future work will be directed to scenarios where synchronization within a multicast group has been lost. A second line of work is to speed up coin generation by the customer smart card and coin validity check by routers: this would require replacing the discrete exponentiation with a faster homomorphic one-way function.

6 Acknowledgments

This work has been partly supported by the European Commission under project IST-2001-32012 "Co-Orthogonal Codes" and by the Spanish Ministry of Science and Technology and the European FEDER Fund under project TIC2001-0633-C03-01 "STREAMOBILE".

References

[1] G. Caronni, K. Waldvogel, D. Sun and B. Plattner, "Efficient security for large and dynamic multicast groups", in *IEEE 7th Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises* (WET ICE '98). Los Alamitos CA: IEEE Computer Society, pp. 376-383, 1998.

[2] S. Deering, D. Estrin, D. Farinacci, V. Jacobson, C. Liu and L. Wei, "The PIM architecture for wide-area multicast routing", *IEEE/ACM Transactions on Networking*, vol. 4, no. 2, pp. 153-162, Apr. 1996.

[3] S. Deering, D. Estrin, D. Farinacci, V. Jacobson, A. Helmy, D. Meyer and L. Wei, "Protocol independent multicast version 2 dense mode specification", IETF Internet Draft, Nov. 1998. http://www.ietf.org

[4] J. Domingo-Ferrer, M. Alba and F. Sebé, "Asynchronous large-scale certification based on certificate verification trees", in *Communications and Multimedia Security'2001*. Norwell MA: Kluwer Academic Publishers, pp. 185-196, 2001.

[5] C. K. Miller, *Multicast Newtorking and Applications*. Reading MA: Addison Wesley, 1999.

[6] J. Moy, "Multicast extensions to OSPF", Internet RFC 1584, March 1994. http://www.ietf.org

[7] R. Oppliger, *Security Technologies for the World Wide Web*. Norwood MA: Artech House, 2000.

[8] B. Quinn and K. Almeroth, "IP multicast applications: challenges and solutions", Internet RFC 3170, Sept. 2001. http://www.ietf.org.

[9] R. Rivest and A. Shamir, "PayWord and Micromint: Two simple micropayment schemes", Technical Report, MIT LCS, Nov. 1995.

[10] R. L. Rivest, A. Shamir and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems", *Communications of the ACM*, vol. 21, pp. 120-126, Feb. 1978.

[11] B. Schneier, *Applied Cryptography*. New York: Wiley, 1996.

[12] J. Snoeyink, S. Suri and G. Varghese, "A lower bound for multicast key distribution", in *Proceedings of IEEE INFOCOM 2001*. Piscataway NJ: IEEE Computer and Communications Society, pp. 422-431, 2001.

[13] D. R. Stinson, *Cryptography. Theory and Practice*, Boca Raton FL: CRC Press, 1995.

[14] R. Wittmann and M. Zitterbart, *Multicast Communication, Protocols and Applications.* San Mateo CA: Morgan Kaufmann, 2001.

[15] *Digital Signature Standard*, FIPS PUB 186, National Institute of Standards, Washington D. C., May 1994.

On the implementation of the Advanced Encryption Standard on a public-key crypto-coprocessor

Antonio Valverde Garcia, Jean-Pierre Seifert

Infineon Technologies, Security & ChipCard ICs
{antonio.valverde,jean-pierre.seifert}@infineon.com

Abstract

This paper describes how to implement the new Advanced Encryption Standard (AES) using a modular arithmetic crypto-coprocessor, typically used to speed up public-key crypto-systems. This idea provides a fast and secure AES implementation when a dedicated hardware AES module is not available. The advantages of using the modular arithmetic coprocessor when compared to a pure software implementation are:

- much higher execution performance,
- less memory usage, and
- optimized protection against side-channel attacks.

Keywords: AES, Crypto-Coprocessor, Implementation Issues, Secure Implementation.

1 Introduction

The Advanced Encryption Standard (AES) specifies a FIPS-approved (cf. [FIPS]) cryptographic algorithm that is used to safely protect electronic data. The AES algorithm is a symmetric block cipher that is able to encrypt (encipher) and decrypt (decipher) electronic data. The AES algorithm is capable of using cryptographic keys of 128, 192, and 256 bits to encrypt and decrypt data blocks of 128 bits. The new AES (also known as Rijndael, cf. [DR2]) is an algorithm designed to use only single byte operations. Therefore, it is an algorithm very suitable for 8-bit μ-processors with only a few kB RAM as commonly used in todays smart cards. However, Rijndael is also well suited for 32-bit μ-processors with more RAM and clearly for dedicated hardware implementations, cf. [Wo, WOL, SMTM]. An optimized implementation of the AES algorithm on an 8051 based μ-controller with a 128-bit key takes less than 1ms @ 15MHz and requires 48 bytes of directly addressable internal RAM to encrypt a 128 bit data block and a little bit more time to decrypt it. Even if this is enough for a large variety of applications, there are some others where the bit rate achieved with this implementation may not be enough (for instance in a contactless environment) or, there is a demand for a high physical attack resistancy. On the other hand, dedicated public-key coprocessors are fast arithmetic coprocessors that usually can handle non-modular and especially modular arithmetic on prime fields $\mathbb{F}_p$ and especially on fields of characteristic two $\mathbb{F}_2^d$, cf. [NR]. These coprocessors are designed to be very efficient for RSA and ECC algorithms, but they are clearly not intended to accelerate the computation of symmetric key algorithms like DES or AES. However, some of the operations usually implemented in a modular arithmetic coprocessor, specifically in those intended for elliptic curve cryptography, are still useful to implement the AES because some transformations of the AES are performed on a field $\mathbb{F}_2^d$. By performing these transformations within the coprocessor, we can reduce the execution time of the encryption and decryption algorithms, reduce the usage of internal RAM memory and protect the algorithm against various side-channel attacks [A, AK1, AK2, CJRR, CKN, DR1, DPV, Gu1, Gu2, KK, Koca], such like timing attacks [KQ, Koch], power attacks [AG, BS99, CCD, KJJ, Me], electromagnetic radiation attacks [SQ] or even fault attacks [ABFHS, BDL, BDHJNT, BS97, BS02, BMM, JLQ, JPY, JQBD, JQYY, KR, KWMK, Ma, Pai, SA, YJ, YKLM1, YKLM2, ZM].

Although many implementations of Rijndael have been brought into the literature, since this algorithm has won the AES contest, none of these implementations so far used a public-key cryptocoprocessor. Therefore, we cannot compare our implementation with any other, and we recommend to look at cf. [Li] to get an overview of alternative implementations on other platforms.

In the course of this paper we first give some hints of the utility of our implementation in many smart card applications. In the next chapter we describe the minimum requirements for the needed coprocessor and give an example of its required architecture. Hereafter, we briefly describe the AES itself. The following chapter is the most important one, as it describes our proposed implementation technique used for the AES. Finally, some security considerations are discussed around the implementation presented here and some estimation figures on the performance of the implementation are also given.

2 Applications

2.1 Chipcard ICs

Chipcards are mainly used to identify and authenticate a card user to a system. The identification or authentication protocol is normally based on symmetric and asymmetric cryptography. Moreover, all the data transfers between the Chipcard and the Terminal are usually protected by a Message Authentication Code (MAC) calculated with a symmetric algorithm. Triple DES is the most currently symmetric algorithm used today in smart cards. However, the new encryption standard (AES) will progressively replace the Triple DES within the next years. Thus, a very efficient AES implementation will be required in those environments where the transaction time is required to be as short as possible, as in the case of contactless applications.

2.2 Security ICs

In the area of Security ICs, like a Trusted Platform Module or a SmartUSB μ-controller, the use of a modular arithmetic coprocessor for the AES implementation described here, will provide an encryption engine, fast enough and very secure for many applications, like bulk encryption, that with a standard software implementation could not be achieved.

2.3 Secure Storage ICs

The main product that can benefit of the AES implementation described here is the so called multimedia card also known as a secure storage IC. This card is typically composed of a large fiash memory, a fast I/O interface and some security logic. When a small CPU and a modular arithmetic coprocessor is incorporated, the AES implementation described here will provide new features like data encryption and decryption which will allow to build new applications like fast and secure memory personalization. This kind of applications require a fast encryption/decryption engine, as fast as the I/O interface to avoid a penalty during the execution time of the application.

2.4 The required modular arithmetic coprocessor

The modular arithmetic coprocessor must have at least 6 registers (4 if only encryption is implemented), each of length greater or equal than 16 bytes each. On the other hand, the coprocessor shall be able to perform the following arithmetic and logical operations:

- Multiplication in $\mathbb{F}_2^d$, $d \geq 128$, of a long register by an 8-bit value,
- Addition modulo 2 ($\oplus$, i.e. exclusive OR) of two long registers,
- Right and Left shifting of a long register,
- Logical AND of two long registers (optional),
- Simultaneous rotation of 4 bytes words (optional).

The following figure gives an example how such a coprocessor could look like: Here, it is supposed that the standard CPU can directly operate on the data

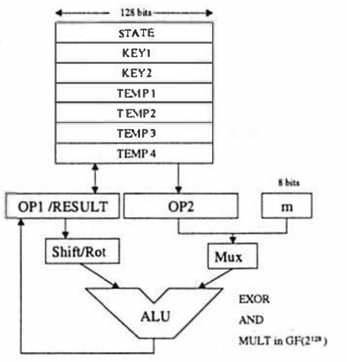


Figure 1: Example of the copprocessor's architecture.

stored on the coprocessors registers but that operations on these registers are much less efficient than on the standard CPU internal registers, because the data stored in those registers have to be transferred to the CPU through some external bus, as these data are usually organized as a so called XRAM.

3 Description of the Advanced Encryption Standard

In this section we briefly describe the Advanced Encryption Standard (AES). For a more detailed description we refer to [DR2].

AES encrypts plaintexts consisting of lb bytes, where $\mathtt{lb} = 16, 24$, or 32. The plaintext is organized as a $(4 \times \mathtt{Nb})$ array (a_{ij}), $0 \le i < 4, 0 \le j < \mathtt{Nb} - 1$, where $\mathtt{Nb} = 4, 6, 8$, depending on the value of lb. The n-th byte of the plaintext is stored in byte $a_{i,j}$ with $i = n \mod 4$, $j = \lfloor \frac{n}{4} \rfloor$.

AES uses a secret key, called *cipher key*, consisting of lk bytes, where $\mathtt{lk} = 16, 24$, or 32. Any combination of values lb and lk is allowed. The cipher key is organized in a $4 \times \mathtt{Nk}$ array (k_{ij}), $0 \le i < 4, 0 \le j \le \mathtt{Nk} - 1$, where $\mathtt{Nk} = 4, 6, 8$, depending on the value of lk. The n-th key byte is stored in byte k_{ij} with $i = n \mod 4$, $j = \lfloor \frac{n}{4} \rfloor$.

The AES encryption process is composed of *rounds*. Except for the last round, each round consists of four transformations called ByteSub, ShiftRow, MixColumn, and AddRoundKey. In the last round the transformation MixColumn is omitted. The four transformations operate on intermediate results, called *states*. A state is a $4 \times \mathtt{Nb}$ array (a_{ij}) of bytes. Initially, the state is given by the plaintext to be encrypted. The number of rounds Nr is 10, 12, or 14, depending on $\max\{\mathtt{Nb}, \mathtt{Nk}\}$. In addition to the transformations performed in the Nr rounds there is an AddRoundKey applied to the plaintext prior to the first round. We call this the *initial* AddRoundKey.

Next, we are going to describe the transformations used in the AES encryption process. We begin with AddRoundKey.

The transformation AddRoundKey The input to the transformation AddRoundKey is a state (a_{ij}), $0 \le i < 4, 0 \le j < \mathtt{Nb}$, and a *round key*, which is an array of bytes $(\mathtt{rk}_{ij})$, $0 \le i < 4, 0 \le j < \mathtt{Nb}$. The output of AddRoundKey is the state $(b_{ij}), 0 \le i < 4, 0 \le j < \mathtt{Nb}$, where

$$b_{ij} = a_{ij} \oplus \mathtt{rk}_{ij}.$$

The round keys are obtained from the cipher key by expanding the cipher key array (k_{ij}) into an array (k_{ij}), $0 \le i < 4, 0 \le j \le \mathtt{Nr} \cdot \mathtt{Nb}$, called the *expanded key*. The round key for the initial application of AddRoundKey is given by the first Nb columns of the expanded key. The round key for the application of AddRoundKey in the m-th round of AES is given by columns $m\mathtt{Nb}, \ldots, (m+1)\mathtt{Nb} - 1$ of the expanded key, $1 \le m \le \mathtt{Nr}$.

The transformation ByteSub Given a state (a_{ij}), $0 \le i < 4, 0 \le j < \mathtt{Nb}$, the transformation ByteSub applies an invertible function $S : \{0,1\}^8 \to \{0,1\}^8$ to each state byte a_{ij} separately. The exact nature of S is of no relevance for the implementation described later. We just mention that S is non-linear, and in fact, it is the only non-linear part of the AES encryption process. In practice, S is often realized by a substitution table or *S-box*.

The transformation ShiftRow The transformation ShiftRow cyclically shifts each row of a state (a_{ij}) separately to the left. Row 0 is not shifted. Rows 1, 2, 3 are shifted by C_1, C_2, C_3 bytes, respectively, where the values of the C_i depend on Nb.

The transformation `MixColumn` The transformation `MixColumn` is crucial to the kind of our special implementation. The transformation `MixColumn` operates on the columns of a state separately. To each column a fixed linear transformation is applied. To do so, bytes are interpreted as elements in the field $\mathbb{F}_{2^8}$. As is usually done, we will denote elements in this field in hexadecimal notation. Hence 01, 02 and 03 correspond to the bytes 00000001, 00000010, and 00000011, respectively. Now `MixColumn` applies to each row of a state the linear transformation defined by the following matrix

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix}. \tag{1}$$

One complete round of the AES encryption procedure is schematically shown in figure 2.

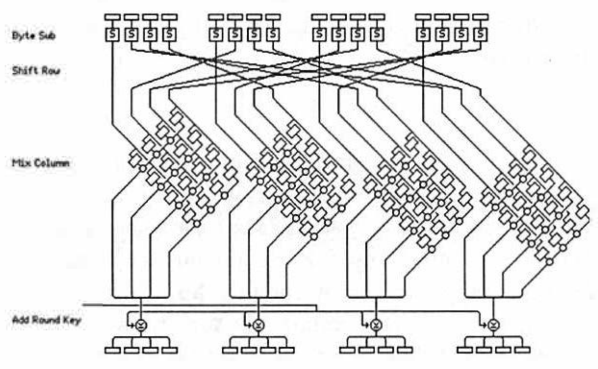


Figure 2: AES round description, cf. [Sa].

The operation `xtime` The multiplications in $\mathbb{F}_{2^8}$ necessary to compute the transformation `MixColumn` are of great importance to our implementation. Therefore we are going to describe them in more detail. First we need to say a few words about the representation of the field $\mathbb{F}_{2^8}$. In AES the field $\mathbb{F}_{2^8}$ is represented as

$$\mathbb{F}_{2^8} = \mathbb{F}_2[x]/(x^8 + x^4 + x^3 + x + 1). \tag{2}$$

That is, elements of $\mathbb{F}_{2^8}$ are polynomials over $\mathbb{F}_2$ of degree at most 7. The addition and multiplication of two polynomials is done modulo the polynomial $x^8 + x^4 + x^3 + x + 1$. Since this is an irreducible polynomial over $\mathbb{F}_2$, (2) defines a field. In this representation of $\mathbb{F}_{2^8}$ the byte $\mathbf{a} = (a_7, \dots, a_1, a_0)$ corresponds to the polynomial $a_7x^7 + \cdots a_1x + a_0$. The multiplication of an element $\mathbf{a} = (a_7, \dots, a_1, a_0)$ in $\mathbb{F}_{2^8}$ by 01, 02, and 03 is realized by multiplying the polynomial $a_7x^7 + \cdots a_1x + a_0$ with the polynomials $1, x, x+1$, respectively, and reducing the result modulo $x^8 + x^4 + x^3 + x + 1$. Hence

$$\begin{aligned} 01 \cdot \mathbf{a} &= \mathbf{a} \\ 03 \cdot \mathbf{a} &= 02 \cdot \mathbf{a} + \mathbf{a}. \end{aligned}$$

We see that the only non-trivial multiplication needed to multiply a column of a state by the matrix in (1) is the multiplication by 02. Following the notation in [DR2] we denote the multiplication of byte $\mathbf{a}$ by 02 by `xtime`(a). The crucial observation is that `xtime`(a) is simply a shift of byte a, followed in some cases by an xor of two bytes. More precisely, for $\mathbf{a} = (a_7, \dots, a_0)$

$$\texttt{xtime}(\mathbf{a}) = \begin{cases} (a_6, \dots, a_0, 0), \\ \quad \text{if } a_7 = 0 \\ \\ (a_6, \dots, a_0, 0) \oplus (0,0,0,1,1,0,1,1), \\ \quad \text{if } a_7 = 1 \end{cases}$$

This finishes our brief description of the AES encryption procedure.

In a pure software implementation of the algorithm on an 8051 based μ-controller these transformations are performed one after the other within the CPU using 48 bytes of directly addressable internal RAM, and taking roughly 12000 clock cycles to encrypt a 128 bit data block with a 128-bit key. The decryption algorithm takes about 30% more time than the cipher and requires at least the same bytes of internal RAM resources. This is due to the fact that the software implementation of the inverse `MixColumn` transformation used for decryption is less efficient than the `MixColumn` transformation used for encryption.

4 The public-key coprocessor based AES implementation

The formerly mentioned type of public-key coprocessor is actually useful to improve the performance of the following transformations of the AES cipher:

- `MixColumn`,
- inverse `MixColumn`,

- KeyExpansion and
- AddRoundKey.

Other transformations like the ByteSub and ShiftRow are performed inside the standard CPU and therefore remain unchanged. The reason of not using the coprocessor to accelerate these two last transformations is the following. The fastest way of performing the ByteSub transformation is by the use of a look-up table (the so called S-Box) containing 256 8-bit values. Because both of them, table indices and table contents are 8-bit values, the 8-bit CPU is the most suitable unit to perform this table access. Nevertheless, we advice the reader to carefully consult our section 5 on the physical security of the AES.

On the other hand, the ShiftRow transformation can be embedded into the ByteSub transformation in such a way that there is no performance loss. The next figure describes the execution parts executed in the CPU and the other ones executed within the coprocessor:

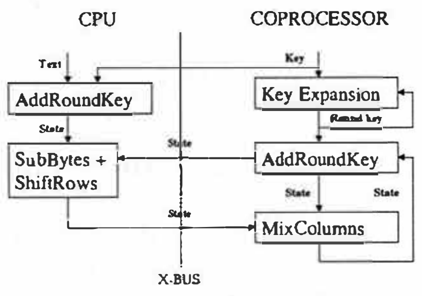


Figure 3: Execution of the AES transformations .

4.1 The MixColumn transformation

The multiplication of columns (MixColumn) is based on the xtime operation as defined within the AES specification. It multiplies a byte of the so called state by 2 modulo the irreducible polynomial $x^8 + x^4 + x^3 + x + 1$. This operation is usually performed on a byte by left shifting the byte (multiplication by 2) and, in case of overflow, xoring (addition modulo 2) with the hexadecimal value $0x1b$.

The MixColumn transformation requires matrix multiplication in the field $\mathbb{F}_2^8$. In an 8-bit CPU, this can be implemented in an efficient way for each column as follows:

$$\begin{aligned} y_0 &= 02 * x_0 \oplus 03 * x_1 \oplus 01 * x_2 \oplus 01 * x_3 \\ y_1 &= 01 * x_0 \oplus 02 * x_1 \oplus 03 * x_2 \oplus 01 * x_3 \\ y_2 &= 01 * x_0 \oplus 01 * x_1 \oplus 02 * x_2 \oplus 03 * x_3 \\ y_3 &= 03 * x_0 \oplus 01 * x_1 \oplus 01 * x_2 \oplus 02 * x_3, \end{aligned}$$

where $*$ represents the xtime operation. After reordering the equations we get:

$$\begin{aligned} y_0 &= 02 * x_0 \oplus 03 * x_1 \oplus x_2 \oplus x3 \\ y_1 &= 02 * x_1 \oplus 03 * x_2 \oplus x_3 \oplus x0 \\ y_2 &= 02 * x_2 \oplus 03 * x_3 \oplus x_0 \oplus x1 \\ y_3 &= 02 * x_3 \oplus 03 * x_0 \oplus x_1 \oplus x2 \end{aligned}$$

The xtime operation can be performed inside the coprocessor on the 16 bytes of the state in parallel via the following formula:

$$\begin{aligned} \texttt{xtime}(\text{state}) &= ((\text{state}\&m_2) << 1) \oplus \\ &\quad (((\text{state}\&m_1) >> 7) * m_3), \end{aligned}$$

where $m_1 = 0x8080...80$ (16 bytes), $m_2 = 0x7f7f...7f$ (16 bytes) and $m_3 = 0x1b$. Here, $*$ denotes the multiplication operation in $\mathbb{F}_2^8$, $\oplus$ is the addition modulo 2, & the AND operation and $<<$ and $>>$ are the bit-left and bit-right shift operations respectively.

The xtime operation itself can be implemented inside the coprocessor with only two temporary registers, as shown below:

$$\begin{aligned} t_1 &= \text{state}\&m_1 \\ t_1 &= t_1 >> 7 \\ t_1 &= t_1 * m_3 \\ t_2 &= \text{state}\&m_2 \\ t_2 &= t_2 << 1 \\ t_1 &= t_1 \oplus t_2 \end{aligned}$$

If the AND operation is not supported by the coprocessor, it has to be done in the standard CPU before loading the state into the coprocessor's register. Then, one has to load the result of the AND operations in both t_1 and t_2. Based on the previous definition of the xtime operation, the whole MixColumn transformation can be defined to operate on the 16 bytes of the state in parallel. The implementation is based on the previous definition of the `xtime` operation:

$$\begin{aligned}
t_1 &= \texttt{xtime}(\text{state}) \\
t_2 &= t_1 \oplus \text{state} \\
t_2 &= \texttt{RotWord}(t_2) \\
t_1 &= t_1 \oplus t_2 \\
t_2 &= \texttt{RotWord}(\text{state}) \\
t_2 &= \texttt{RotWord}(t_2) \\
t_1 &= t_1 \oplus t_2 \\
t_2 &= \texttt{RotWord}(t_2) \\
\text{state} &= t_1 \oplus t_2
\end{aligned}$$

The total number of registers needed for the implementation of the `MixColumn` transformation in the coprocessor is 3, two temporal registers for the intermediate results and another for the state.

The `RotWord` operation as defined in the AES specification has to be performed on every 4 bytes of the state independently. If it is not supported by the coprocessor, this operation must be done by the standard CPU, accessing the internal coprocessor's registers.

4.2 The inverse `MixColumn` transformation

The inverse `MixColumn` transformation requires also a matrix multiplication in the field $\mathbb{F}_2^8$. In an 8-bit CPU, this can be implemented in an efficient way for each column as follows:

$$\begin{aligned}
y_0 &= 0e * x_0 \oplus 0b * x_1 \oplus 0d * x_2 \oplus 09 * x_3 \\
y_1 &= 09 * x_0 \oplus 0e * x_1 \oplus 0b * x_2 \oplus 0d * x_3 \\
y_2 &= 0d * x_0 \oplus 09 * x_1 \oplus 0e * x_2 \oplus 0b * x_3 \\
y_3 &= 0b * x_0 \oplus 0d * x_1 \oplus 09 * x_2 \oplus 0e * x_3.
\end{aligned}$$

After reordering the equations we get:

$$\begin{aligned}
y_0 &= 0e * x_0 \oplus 0b * x_1 \oplus 0d * x_2 \oplus 09 * x_3 \\
y_1 &= 0e * x_1 \oplus 0b * x_2 \oplus 0d * x_3 \oplus 09 * x_0 \\
y_2 &= 0e * x_2 \oplus 0b * x_3 \oplus 0d * x_0 \oplus 09 * x_1 \\
y_3 &= 0e * x_3 \oplus 0b * x_0 \oplus 0d * x_1 \oplus 09 * x_2.
\end{aligned}$$

As for the `MixColumn`, the inverse transformation (needed for decryption) can also be defined to operate on the 16 bytes of the state in parallel. The implementation is based on the previous definition of the `xtime` operation:

$$\begin{aligned}
t_1 &= \texttt{xtime}(\text{state}) \\
t_2 &= \texttt{xtime}(t_1) \\
t_3 &= \texttt{xtime}(t_2) \\
t_4 &= t_1 \oplus t_2 \oplus t_3 \\
t_2 &= \text{state} \oplus t_2 \oplus t_3 \\
t_1 &= \text{state} \oplus t_1 \oplus t_3 \\
t_3 &= \text{state} \oplus t_3 \\
t_1 &= \texttt{RotWord}(t_1) \\
t_2 &= \texttt{RotWord}(\texttt{RotWord}(t_2)) \\
t_3 &= \texttt{RotWord}(\texttt{RotWord}(\texttt{RotWord}(t_3))) \\
\text{state} &= t_1 \oplus t_2 \oplus t_3 \oplus t_4
\end{aligned}$$

The total number of registers needed for the implementation of the inverse transformation in the coprocessor is 5, where 4 temporal registers are used for intermediate results and one other register for the state itself.

Another way to implement the inverse `MixColumn` transformation is by definition of the following two new operations:

$$\begin{aligned}
\texttt{xtime}_4(\text{state}) &= ((\text{state}\&m_5) << 2) \oplus \\
&\quad (((\text{state}\&m_4) >> 6) * m_3) \\
\texttt{xtime}_8(\text{state}) &= ((\text{state}\&m_7) << 3) \oplus \\
&\quad (((\text{state}\&m_6) >> 5) * m_3),
\end{aligned}$$

where $m_4 = 0xc0c0...c0$ (16 bytes), $m_5 = 0x3f3f...3f$ (16 bytes), $m_6 = 0xe0e0...e0$ (16 bytes), $m_7 = 0x1f1f...1f$ (16 bytes) and $m_3 = 0x1b$. Therefore, the implementation of the inverse `MixColumn` transformation can be redefined as follows:

$$\begin{aligned}
t_1 &= \texttt{xtime}(\text{state}) \\
t_2 &= \texttt{xtime}_4(\text{state}) \\
t_3 &= \texttt{xtime}_8(\text{state}) \\
t_4 &= t_1 \oplus t_2 \oplus t_3 \\
t_2 &= \text{state} \oplus t_2 \oplus t_3 \\
t_1 &= \text{state} \oplus t_1 \oplus t_3 \\
t_3 &= \text{state} \oplus t_3 \\
t_1 &= \texttt{RotWord}(t_1) \\
t_2 &= \texttt{RotWord}(\texttt{RotWord}(t_2)) \\
t_3 &= \texttt{RotWord}(\texttt{RotWord}(\texttt{RotWord}(t_3))) \\
\text{state} &= t_1 \oplus t_2 \oplus t_3 \oplus t_4
\end{aligned}$$

The advantage of this second implementation is that the operations `xtime`, $\texttt{xtime}_4$ and $\texttt{xtime}_8$ can be calculated in parallel from the state, avoiding the sequence of the first implementation. M oreover, in the case that the AND operation is not available within the coprocessor, this second solution allows to precompute all the AND values within the standard CPU before loading the state into the coprocessor.

4.3 The Key Expansion

The 16, 24 or 32 bytes of the key (depending on the key length) are loaded into the Key register[1] of the coprocessor (Key1 and Key2 registers for 256-bit keys). Then, the next round key bytes are calculated with the following sequence of operations.

For a 128-bit key, perform the following sequence, and for each intermediate round do:

$$\begin{aligned}
t_1 &= \mathtt{Rcon} \oplus \mathtt{ByteSub}(\mathtt{RotWord}(\mathrm{Key}))\\
\mathrm{Key} &= \mathrm{Key} \oplus t_1\\
t_1 &= \mathrm{Key}\\
t_1 &= t_1 >> 32\\
\mathrm{Key} &= \mathrm{Key} \oplus t_1\\
t_1 &= t_1 >> 32\\
\mathrm{Key} &= \mathrm{Key} \oplus t_1\\
t_1 &= t_1 >> 32\\
\mathrm{Key} &= \mathrm{Key} \oplus t_1.
\end{aligned}$$

The `RotWord`, `ByteSub` operations are performed by the standard CPU on the 4 rightmost bytes of the Key register, then storing the result into the 4 leftmost bytes of t_1 and clearing the other bytes. `Rcon` is the 4-byte constant defined within the AES specification.

For a 256-bit key perform the following sequence, and for each intermediate "even" round do:

$$\begin{aligned}
t_1 &= \mathtt{Rcon} \oplus \mathtt{ByteSub}(\mathtt{RotWord}(\mathrm{Key2}))\\
\mathrm{Key}_1 &= \mathrm{Key}_1 \oplus t_1\\
t_1 &= \mathrm{Key}_1\\
t_1 &= t_1 >> 32\\
\mathrm{Key}_1 &= \mathrm{Key}_1 \oplus t_1\\
t_1 &= t_1 >> 32\\
\mathrm{Key}_1 &= \mathrm{Key}_1 \oplus t_1\\
t_1 &= t_1 >> 32\\
\mathrm{Key}_1 &= \mathrm{Key}_1 \oplus t_1
\end{aligned}$$

while every intermediate "odd" round (except round 1) is done as:

$$\begin{aligned}
t_1 &= \mathtt{ByteSub}(\mathrm{Key1})\\
\mathrm{Key}_2 &= \mathrm{Key}_2 \oplus t_1\\
t_1 &= \mathrm{Key}_2\\
t_1 &= t_1 >> 32\\
\mathrm{Key}_2 &= \mathrm{Key}_2 \oplus t_1\\
t_1 &= t_1 >> 32\\
\mathrm{Key}_2 &= \mathrm{Key}_2 \oplus t_1\\
t_1 &= t_1 >> 32\\
\mathrm{Key}_2 &= \mathrm{Key}_2 \oplus t_1
\end{aligned}$$

For 196-bit keys, the sequence gets more complicated as in that case, new round key bytes are generated within a window of 6 bytes, but round key bytes should be delivered at a rate of 4 bytes. Basically, the process to generate the new round key bytes is similar to that for 128 bit keys, but yet longer registers (24 bytes long) and/or an additional temporary register might be needed.

Totally, the number of registers needed for the implementation of the Key Expansion transformation within the coprocessor is 2 (or at maximum 3 for keys longer than 16 bytes).

4.4 The `AddRoundKey` transformation

This transformation is performed by simply adding the state and the key modulo 2 inside the coprocessor:

$$\mathrm{state} = \mathrm{Key} \oplus \mathrm{state}.$$

No temporal register is therefore needed. The Key register used will be Key1 or Key2 in the case of 256-bit keys, depending on the round number (Key1 for "even" rounds and Key2 for "odd" rounds).

[1] Mapping the encryption or decryption key to the Key register is straightforward, bytes $a_0, a_1, ..., a_{15}$ of the key are mapped one to one to bytes $k_0, k_1, ..., k_{15}$ of the Key register respectively.

5 Security Considerations

Although there is a large variety of possible physical attacks on the AES, cf. [AG, BS99, BS02, CJRR, DR1, KQ, KWMK, Me, YT], the `xtime` operation is clearly the most critical one in the AES algorithm, at least with respect to physical security or so called side-channel attacks. Namely, this operation involves a multiplication that is subject to timing and fault attacks (see [KQ, BS02]). We also stress that the recently developed fault based susceptibility due to [BS02] cannot be avoided by the simple dedicated fault-tolerant AES hardware as proposed by [KWMK].

However, thanks to the implementation described here, the aforesaid timing attack on the `xtime` operation doesnt work. This is due to the fact that the timing behaviour of modern crypto coprocessors is independent of its operands, which indeed avoids a timing attack vulnerability of our implementation.

Moreover, by performing the `xtime` operation on 16 bytes in parallel we make fault attacks very difficult to achieve, because we can use a fault in the calculation to flip a bit, but the flipped bit can be any one of the 128 bits of the state or temporary variable. Another critical part of the implementation described here might be the transfer of data through the so called X-BUS, the bus that connects the CPU and the coprocessor. This transfer of data is more significant when the AND and Rotate operations are not supported by the coprocessor and therefore have to be performed within the standard CPU. The bus contents could then be tampered via an electronic microscope, a focused ion beam, or could be revealed through measuring the power consumption or even by an electromagnetic field analysis.

Fortunately, this X-BUS is by some μ-controller ICs vendors protected by hardware and/or software mechanisms. Among the hardware countermeasures there are active shields or random bus scrambling techniques available on some existing high security m-controller ICs. Last generation of those high securitμ-controller ICs are designed using a special dual rail security logic, cf. [MAK, MACMT]. This logic not only ensures that both, a "0" and a "1" have the same Hamming weight, but also that changes between a logical "0" and a logical "1" are not distinguishable by an adversary.

As software measures some masking and encryption techniques could be applied to the data before being transferred, both in the CPU and in the coprocessor. However, these measures may have a significant impact on the overall performance of the algorithm, which makes the aforesaid hardware countermeasures the practically preferred choice.

6 Performance Estimation

An implementation of the AES encryption algorithm with a key length of 128 bits on Infineons SLE66P (8051 based) security controller family, cf. [Inf2], combined together with Infineons recently developed modular arithmetic coprocessor Spiridon, cf. [Inf1] (which has no AND or `RotWord` operation), is approximately two times faster than an optimized 8051 based implementation, and requires only 16 bytes of internal RAM memory. Most importantly, this implementation greatly benefits from the high physical attack security offered by the Spiridon coprocessor, which will be described in another publication.

However, we expect an implementation using an optimal modular arithmetic coprocessor with all the operations described at the beginning of the present paper by at least a factor of four faster than the implementation on Infineons Spiridon.

7 Acknowledgments

We would like to thank W. Fischer for valuable discussions on the material presented within this paper.

References

[A] R. Anderson, *Security Engineering*, John Wiley & Sons, New York, 2001.

[ABFHS] C. Aumüller, B. Bier, W. Fischer, P. Hofreiter, J.-P. Seifert, "Fault attacks on RSA: Concrete results and practical countermeasures", *Proc. of CHES '02*, Springer LNCS, pp. 261-276, 2002.

[AG] M. L. Akkar, C. Giraud, "An implementation of DES and AES, secure against some attacks", *Proc. of CHES '01*, Springer LNCS vol. 2162, pp. 315-324, 2001.

[AK1] R. Anderson, M. Kuhn, "Tamper Resistance – a cautionary note", *Proc. of 2nd USENIX Workshop on Electronic Commerce*, pp. 1-11, 1996.

[AK2] R. Anderson, M. Kuhn, "Low cost attacks attacks on tamper resistant devices", *Proc. of 1997 Security Protocols Workshop*, Springer LNCS vol. 1361, pp. 125-136, 1997.

[BDL] D. Boneh, R. A. DeMillo, R. Lipton, "On the Importance of Eliminating Errors in Cryptographic Computations" *Journal of Cryptology* **14**(2):101-120, 2001.

[BDHJNT] F. Bao, R. H. Deng, Y. Han, A. Jeng, A. D. Narasimbalu, T. Ngair, "Breaking public key cryptosystems on tamper resistant dives in the presence of transient faults", *Proc. of 1997 Security Protocols Workshop*, Springer LNCS vol. 1361, pp. 115-124, 1997.

[BS97] E. Biham, A. Shamir, "Differential fault analysis of secret key cryptosystems", *Proc. of CRYPTO '97*, Springer LNCS vol. 1294, pp. 513-525, 1997.

[BS99] E. Biham, A. Shamir, "Power analysis of the key scheduling of the AES candidates", *Proc. of the second AES conference*, pp. 115-121, 1999.

[BS02] J. Blömer, J.-P. Seifert, "Fault based cryptanalysis of the AES", e-Print Archive of the IACR, 2002, `http://www.iacr.org/`.

[BMM] I. Biehl, B. Meyer, V. Müller, "Differential fault attacks on elliptic curve cryptosystems", *Proc. of CRYPTO '00*, Springer LNCS vol. 1880, pp. 131-146, 2000.

[CCD] C. Clavier, J.-S. Coron, N. Dabbous, "Differential Power Analysis in the presence of Hardware Countermeasures", *Proc. of CHES '00*, Springer LNCS vol. 1965, pp. 252-263, 2000.

[CJRR] S. Chari, C. Jutla, J. R. Rao, P. J. Rohatgi, "A cautionary note regarding evaluation of AES candidates on smartcards", *Proc. of the second AES conference*, pp. 135-150, 1999.

[CKN] J.-S. Coron, P. Kocher D. Naccache, "Statistics and Secret Leakage", *Proc. of Financial Cryptography*, Springer LNCS, 2000.

[DR1] J. Daemen, V. Rijmen, "Resistance against implementation attacks: a comparative study", *Proc. of the second AES conference*, pp. 122-132, 1999.

[DR2] J. Daemen, V. Rijmen, *The Design of Rijndael*, Springer-Verlag, Berlin, 2002.

[DPV] J. Daemen, M. Peeters, G. Van Assche, "Bitslice ciphers and implementation attacks", *Proc. of Fast Software Encryption 2000*, Springer LNCS vol. 1978, pp. 134-149, 2001.

[FIPS] Federal Information Processing Standard, "Advanced Encryption Standard (AES)", National Institute of Standards and Technology (NIST) 2001, `http://csrc.nist.gov/publications/drafts/dfips-AES.pdf`.

[Gu1] P. Gutmann, "Secure deletion of data from magnetic and solid-state memory", *Proc. of 6th USENIX Security Symposium*, pp. 77-89, 1997.

[Gu2] P. Gutmann, "Data Remanence in Semiconductor Devices", *Proc. of 7th USENIX Security Symposium*, 1998.

[Inf1] Infineon Technologies AG, "Security & Chip Card ICs, Crypto2000, Modular Arithmetic Coprocessor, Preliminary Confidential Architecture Specification", v1.1, January 2001.

[Inf2] Infineon Technologies AG, "Security & Chip Card ICs, SLE 66Cxxx, Security Controller Family, Preliminary Confidential Data Book", September 2001.

[JLQ] M. Joye, A. K. Lenstra, J.-J. Quisquater, "Chinese remaindering based cryptosystem in the presence of faults", *Journal of Cryptology* **12**(4):241-245, 1999.

[JPY] M. Joye, P. Paillier, S.-M. Yen, "Secure Evaluation of Modular Functions", *Proc. of 2001 International Workshop on Cryptology and Network Security*, pp. 227-229, 2001.

[JQBD] M. Joye, J.-J. Quisquater, F. Bao, R. H. Deng, "RSA-type signatures in the presence of transient faults", *Cryptography and Coding*, Springer LNCS vol. 1335, pp. 155-160, 1997.

[JQYY] M. Joye, J.-J. Quisquater, S. M. Yen, M. Yung, "Observability analysis — detecting when improved cryptosystems fail", *Proc. of CT-RSA Conference 2002*, Springer LNCS vol. 2271, pp. 17-29, 2002.

[KR] B. Kaliski, M. J. B. Robshaw, "Comments on some new attacks on cryptographic devices", *RSA Laboratories Bulletin* **5**, July 1997.

[KK] O. Kömmerling, M. Kuhn, "Design Principles for Tamper-Resistant Smartcard Processors", *Proc. of the USENIX Workshop on Smartcard Technologies*, pp. 9-20, 1999.

[KQ] F. Koeune, J.-J. Quisquater, "A timing attack against Rijndael", *Université catholique de Louvain*, TR CG-1999/1, 6 pages , 1999.

[Koca] O. Kocar, "Hardwaresicherheit von Mikrochips in Chipkarten", *Datenschutz und Datensicherheit* **20**(7):421-424, 1996.

[Koch] P. Kocher, "Timing attacks on implementations of Diffie-Hellmann, RSA, DSS and other systems", *Proc. of CYRPTO '97*, Springer LNCS vol. 1109, pp. 104-113, 1997.

[KJJ] P. Kocher, J. Jaffe, J. Jun, "Differential Power Analysis", *Proc. of CYRPTO '99*, Springer LNCS vol. 1666, pp. 388-397, 1999.

[KWMK] R. Karri, K. Wu, P. Mishra, Y. Kim, "Concurrent error detection of fault-based side-channel cryptanalysis of 128-bit symmetric block ciphers", *Proc. of IEEE Design Automation Conference*, pp. 579-585, 2001.

[Li] H. Lipmaa, "AES candidates, a survey of implementations", `http://www.tcs.hut.fi/~helger/aes/rijndael.html`.

[Ma] D. P. Maher, "Fault induction attacks, tamper resistance, and hostile reverse engineering in perspective", *Proc. of Financial Cryptography*, Springer LNCS vol. 1318, pp. 109-121, 1997.

[Me] T. Messerges, "Securing the AES finalists against power analysis attacks", *Proc. of Fast Software Encryption 2000*, Springer LNCS vol. 1978, pp. 150-164, 2001.

[MAK] S. W. Moore, R. J. Anderson, M. G. Kuhn, "Improving Smartcard Security using Self-Timed Circuit Technology", *Fourth AciD-WG Workshop*, Grenoble, ISBN 2-913329-44-6, 2000.

[MACMT] S. W. Moore, R. J. Anderson, P. Cunningham, R. Mullins, G. Taylor, "Improving Smartcard Security using Self-Timed Circuit Technology", *Proc. of Asynch 2002*, IEEE Computer Society Press, 2002.

[NR] D. Naccache, D. M'Raihi, "Cryptographic smart cards", *IEEE Micro*, pp. 14-24, 1996.

[Pai] P. Pailler, "Evaluating differential fault analysis of unknown cryptosystems", *Gemplus Corporate Product R&D Division*, TR AP05-1998, 8 pages, 1999.

[Pe] I. Petersen, "Chinks in digital armor — Exploiting faults to break smartcard cryptosystems", *Science News* **151**(5):78-79, 1997.

[Sa] J. Savard, "The Advanced Encryption Standard (Rijndael)", `http://home.ecn.ab.ca/~jsavard/crypto/co040801.html`.

[SQ] D. Samyde, J.-J. Quisquater, "ElectroMagnetic Analysis (EMA): Measures and Countermeasures for Smart Cards", *Proc. of Int. Conf. on Research in Smart Cards, E-Smart 2001*, Springer LNCS vol. 2140, pp. 200-210, 2001.

[SMTM] A. Satoh, S. Morioka, K. Takano, S. Munetoh, "A compact Rijndael hardware architecture with S-Box optimization", *Proc. of ASIACRYPT '01*, Springer LNCS, pp. 241-256, 2001.

[SA] S. Skorobogatov, R. Anderson, "Optical Fault Induction Attacks", *Proc. of CHES '02*, Springer LNCS, pp. 2-12, 2002.

[Wo] J. Wolkerstorfer, "An ASIC implementation of the AES MixColumn-operation", Graz University of Technology, Institute for Applied Information Processing and Communications, Manuscript, 4 pages, 2001.

[WOL] J. Wolkerstorfer, E. Oswald, M. Lamberger, "An ASIC implementation of the AES S-Boxes", *Proc. of CT-RSA Conference 2002*, Springer LNCS vol. 2271, 2002.

[YJ] S.-M. Yen, M. Joye, "Checking before output may not be enough against fault-based cryptanalysis", *IEEE Trans. on Computers* **49**:967-970, 2000.

[YKLM1] S.-M. Yen, S.-J. Kim, S.-G. Lim, S.-J. Moon, "RSA Speedup with Residue Number System immune from Hardware fault cryptanalysis", *Proc. of the ICISC 2001*, Springer LNCS, 2001.

[YKLM2] S.-M. Yen, S.-J. Kim, S.-G. Lim, S.-J. Moon, "A countermeasure against one physical cryptanalysis may benefit another attack", *Proc. of the ICISC 2001*, Springer LNCS, 2001.

[YT] S.-M. Yen, S. Y. Tseng, "Differential power cryptanalysis of a Rijndael implementation", LCIS Technical Report TR-2K1-9, Dept. of Computer Science and Information Engineering, National Central University, Taiwan, 2001.

[ZM] Y. Zheng, T. Matsumoto, "Breaking real-world implementations of cryptosystems by manipulating their random number generation", *Proc. of the 1997 Symposium on Cryptography and Information Security*, Springer LNCS, 1997.

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of developers, programmers, system administrators, and architects working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

SAGE, the System Administrators Guild

The System Administrators Guild, a Special Technical Group within the USENIX Association, is dedicated to the recognition and advancement of system administration as a profession. To join SAGE, you must be a member of USENIX.

Member Benefits

- Free subscription to *;login:*, the Association's magazine, published eight times a year, featuring technical articles, system administration tips and techniques, practical columns on Perl, Java, Tcl/Tk, and Open Source, book and software reviews, summaries of sessions at USENIX conferences, and Standards Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts.
- Access to *;login:* on the USENIX Web site.
- Access to papers from the USENIX Conferences and Symposia, starting with 1993, on the USENIX Web site.
- Discounts on registration fees for the annual, multi-topic technical conference, the System Administration Conference (LISA), and the various single-topic symposia addressing topics such as security, Linux, Internet technologies and systems, operating systems, and Windows—as many as twelve technical meetings every year.
- Discounts on the purchase of proceedings and CD-ROMs from USENIX conferences and symposia and other technical publications.
- The right to vote on matters affecting the Association, its bylaws, and election of its directors and officers.
- Savings on a variety of products, books, software, and periodicals: see *http://www.usenix.org/membership/specialdisc.html* for details.

Supporting Members of the USENIX Association

Atos Origin B.V.	Sendmail, Inc.
Freshwater Software	Sun Microsystems, Inc.
Interhack Corporation	Sybase, Inc.
Microsoft Research	Taos: The Sys Admin Company
Motorola Australia Software Centre	UUNET Technologies, Inc.
OSDN	Ximian, Inc.

Supporting Members of SAGE

Certainty Solutions	Microsoft Research
Collective Technologies	OSDN
ESM Services, Inc.	Ripe NCC
Freshwater Software	

For more information about membership, conferences, or publications,
see *http://www.usenix.org/*
or contact:
USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA
Phone: 510-528-8649 Fax: 510-548-5738 Email: *office@usenix.org*

ISBN 1-931971-04-8